IT UNIVERSITY OF COPENHAGEN

DevOps, Software Evolution and Software Maintenance

COURSE CODE: KSDSESM1KU

MSC IN COMPUTER SCIENCE

DevOps: ITU-MiniTwit

GROUP E — GRL PWR

IT UNIVERSITY OF COPENHAGEN

Name	Email
Amalie Bøgild Sørensen	abso@itu.dk
Andreas Nicolaj Tietgen	anti@itu.dk
Malin Birgitta Linnea Nordqvist	bino@itu.dk
Mille Mei Zhen Loo	milo@itu.dk
My Marie Nordal Jensen	myje@itu.dk
Sarah Cecilie Chytræus Christiansen	sacc@itu.dk

14th May 2024

2500 words

Part

Table of Contents

1 System's Perspective

1.1 Design and Architecture

Figure 1 provides an overview of our system infrastructure:

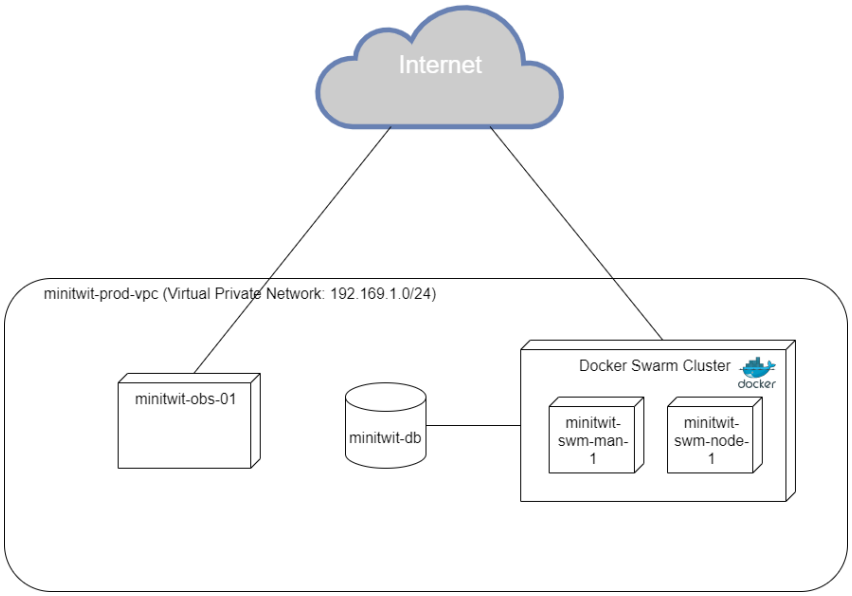

Figure 1: *Overview of the system*

The system includes a single Virtual Private Network with an IP 192.168.1.0/24. Within this network, we operate a database cluster consisting of a single node. This node is connected to the Docker Swarm cluster, featuring both a manager and a worker node.

Both our monitoring and cluster servers are visibly exposed to the internet. Additionally, the database server is accessible from the internet but has constraints to only talk to our servers and therefore not accept requests from any other host.

1.2 Interactions of Subsystems

Our system consists of two main subsystems, namely the REST SimulatorAPI and the webapplication Minitwit. As can be seen in figure 2, The SimulatorAPI and Minitwit subsystems are similar in structure. However when looking at the processes behind the structures in Figure 3 and 4 it is apparant that these two differ a great deal.

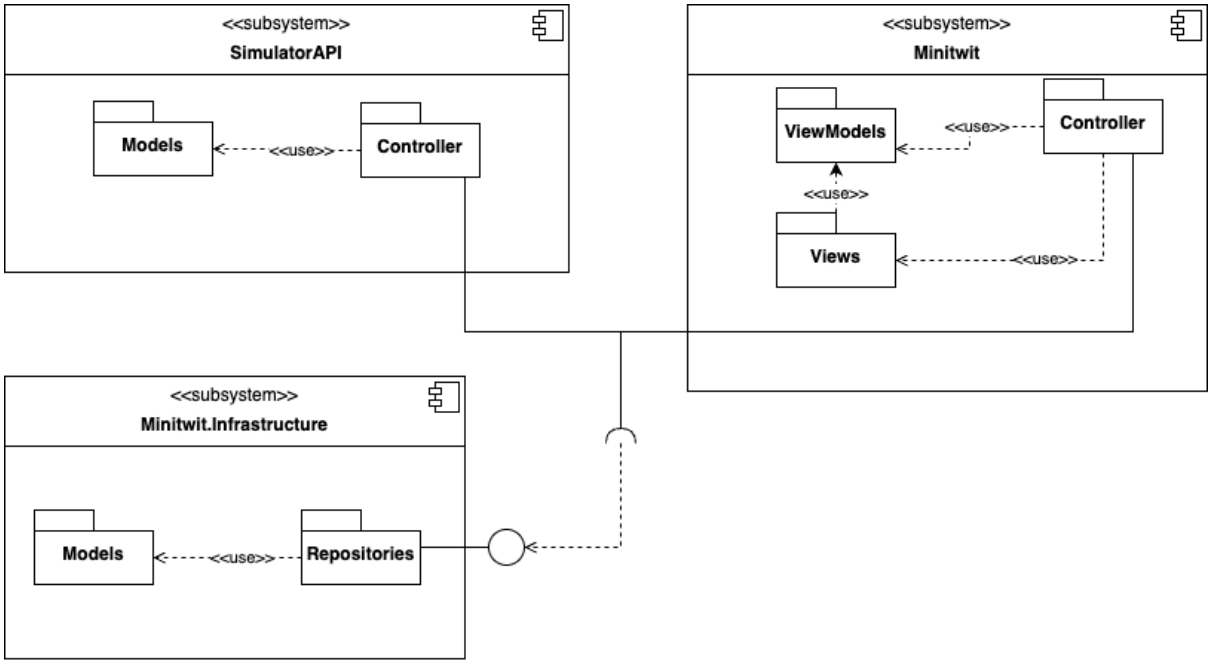

Figure 2: *An overview of the subsystems and how they interact*

Because the two main subsystems are inherently different we wanted to implement a database abstraction layer such that we had a uniform way of querying the database. Therefore, both controllers use the Repository package from the subsystem Minitwit.Infrastructure to query the database using the Repository pattern[5]. Due to the different specifications for the Minitwit and SimulatorAPI, the controller for Minitwit is more complex. As can be seen in figure 3 the controller for Minitwit has to present information in a GUI, whereas the SimulatorAPI only calls the repository to fetch the user messages, as can be seen in figure 4.

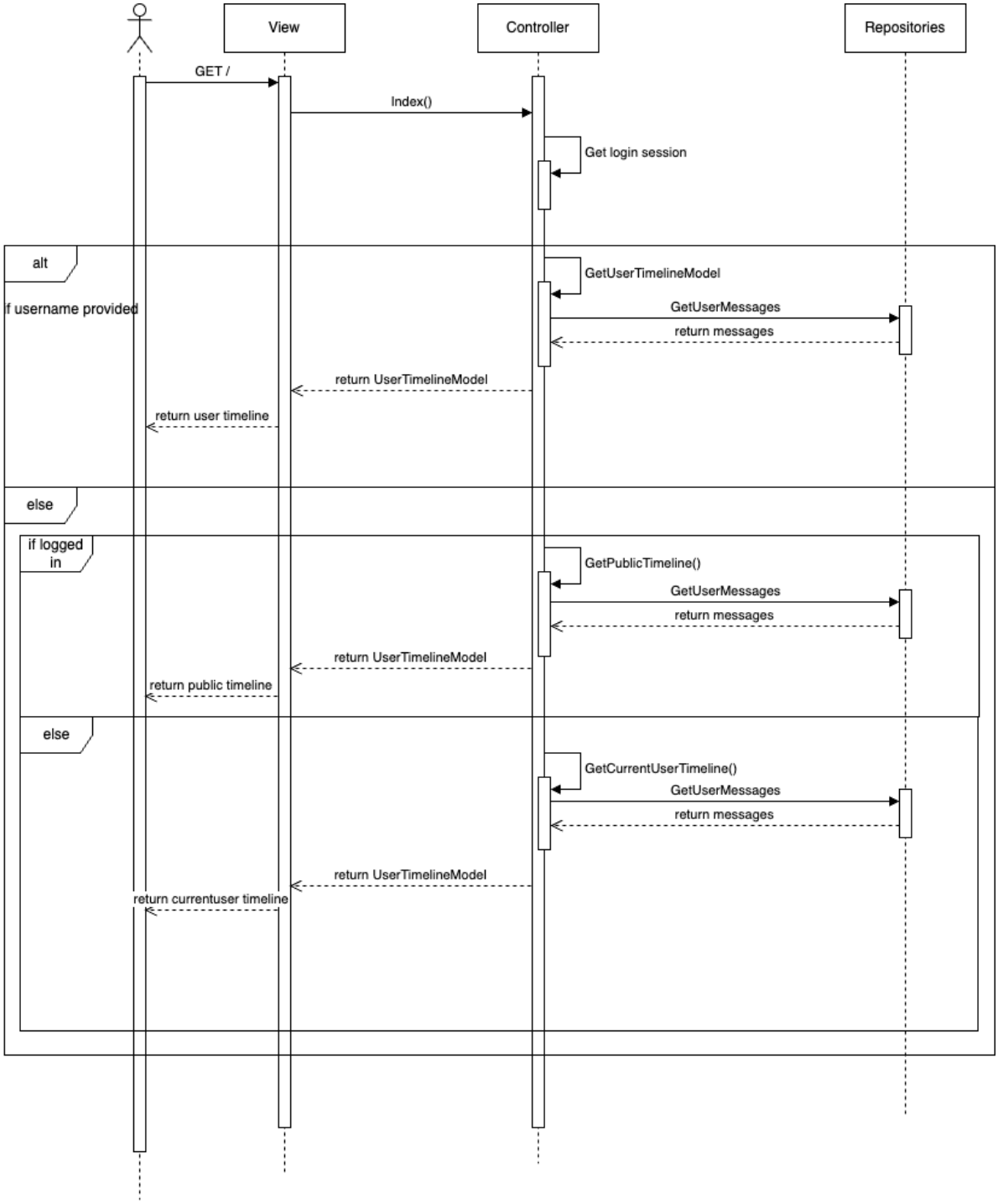

Figure 3: *Minitwit: Sequence diagram of a user call to the index page*

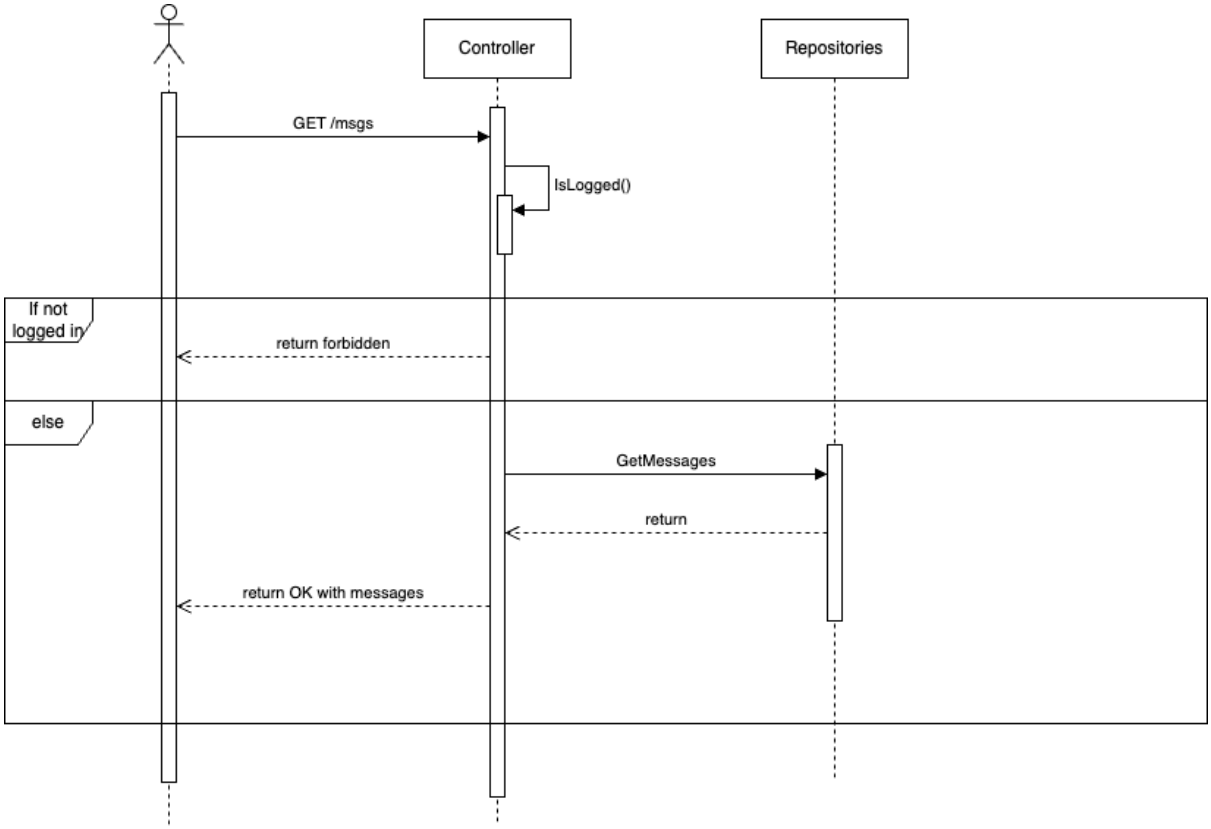

Figure 4: *SimulatorAPI: Sequence diagram of a user call to get messages*

1.3 Dependencies, Technologies and tools

A comprehensive list of tools containing the full reasoning for their use, can be found in appendix C

Purpose	**Technology/Tool**	**Description**
Work log	Notion	Notion is a web hosted workspace, that allows users to collaboratively manage projects, notes, and other types of documents in a markdown format in real-time.

		Initially, we opted to maintain our work log on Github wikis due to it's integration with the repository. However, this approach proved cumbersome as the process of making pull requests and seeking reviews for minor log updates became overly complex. Consequently, we transitioned to Notion.
Programming language	C#	C# is an object-oriented programming language, that supports development for a variety of application types such as web-application. For refactorisation of the application we chose C# , due to it's widespread use in the industry, making it very well documented. Furthermore, the familiarity of object-oriented programming paradigms among developers served as a supplementary factor in the decision
Micro Web-Framework	Razor Pages	Razor pages is a simple web application programming model provided by ASP.NET. It uses C# and simple markup syntax in order to build web pages, and is generally the recommended framework for page-focused, serverside web applications in .NET core. Initially, we attempted using Scriban, due to it's superior efficiency in terms of speed and memory usage, and it's compatibility with html templates. However, due to the lack of online resources, we switched to Razor Pages, which offered simpler configuration and an abundance of documentation, but required more code rebasing.

Database Connection	Entity Framework	Entity Framework serves as a object-relational mapper, which allows developers to work with databases using .NET objects, and eliminates the need for data-access code, by using LINQ in order to interact with the data. Furthermore, EF Core supports a plethora of database engines, such as PostgreSQL and SQLite. We opted for Entity Framework due to the reasons stated above.
Server host	Digital Ocean	Digital Ocean (DO) is a cloud hosting service, which provides server hosting, database hosting, and a command line interface tool (doctl), which can be used in to automatically create and drop droplets (VMs). We have chosen DO primarily due to the fact that it was recommended by the lecturer, but kept using digital ocean, due to the great documentation and online resources.
Database	PostgreSQL	PostgreSQL is an open-source relational database management system known for its extensibility, and adherence to SQL standards. Since the majority of developers in the team have previous experience with PostgreSQL, we chose to migrate from SQLite to PostgreSQL.
Database migration	Digital Ocean	Digital Ocean is a cloud hosting service that provides a database hosting service, which is well documented.

CI/CD	Github Actions	Github actions is a CI/CD service, that allows you to automate tasks such as building, testing, and deploying your code when certain conditions are met. Github Actions is well documented, and it is quite seamless to configure secrets and refer to them in a .yaml file. Due to it's integration with Github, we chose Github Actions, as we already use Github to store our code repository. Furthermore, it collaborates well with other tools used in the project such as Digital Ocean.
Monitoring Graphics	Grafana	Grafana is an open source monitoring toolbox, that includes a variety of features such as visualising monitoring metrics and logging. This is done by configuring dashboards with charts and graphs, which visualise your system's performance and logging outputs. Initially, we opted for OpenTelemetry since it is a widely used metrics exporter and collector. However, due to difficulties with the setup, we switched to Grafana.
Metrics collector	Prometheus	Prometheus is a monitoring and alerting toolkit, that scrapes and stores data from a specific application, alerting developers of abnormalities in the system. The only requirement we had for a metrics collector was: It had to collaborate with Grafana. Furthermore, this tool was recommended by our lecturer.
Metrics collector	OpenTelemetry	OpenTelemetry is a metrics collector, that supports

		We chose OpenTelemetry since it is widely used, and it is well integrated with .NET applications.
Quality of Code Analysis	SonarCloud	SonarCloud is a cloud based code analysis service, which can be integrated into a GitHub repository, in order to analyse the code in said repository. We added this software as it was an exercise in the course.
Linter	Pre-commit	Pre-Commit is a framework for managing pre-commit hooks. This can be used for automatically fixing code format and removing debug statements. We chose this tool as multiple actions can be integrated into one hook. As we were tasked with integrating 3 linters into the system.
Linter	Hadolint	Hadolint is an open-source Dockerfile linter, that enforces best practices when building docker images. Due to the nature of the server setup, we have quite a few Dockerfiles. Therefore, we found it fitting to have a Dockerfile linter. From the alternative presented (Dockerlinter) Hadolint seems to be the more popular one as it is haskell based rather than node.js based.
Linter	Dotnet-Format	Dotnet-Format is a formatting tool, which is included in the .NET 6 SDK and onwards. It applies style preferences to a project which can configured in an .editorconfig file.

		We chose Dotnet-Format since this linter enforces C# coding conventions, and the primary programming language used in this project is C#.
Logging	Grafana	See section on Monitoring Graphics. We chose to use Grafana for logging as we already use Grafana for monitoring. Furthermore, we were interested in keeping the array of tools used in this project from expanding too much.
Logging	Grafana Loki	Loki is a log aggregation system designed to store and query logs from an application inspired by Prometheus. In order to implement a logging stack we were interested in using a tool that was compatible with Grafana. Loki ensures this compatibility, as it is from the Grafana toolbox.
Infrastructure as code	Pulumi	Pulumi is an open-source infrastructure as code SDK, that enables creation, deployment and management of infrastructure for a cloud service. We wanted to move away from using Vagrant to deploy as it lacks collaborative features, making development difficult. We considered 3 alternatives to Vagrant: Terraform, Pulumi, and OpenTofu. In the End we picked Pulumi, as it doesn't introduce a new programming language, it is open-source, and it works well with CI/CD tools.
Configuration management	Ansible	Ansible is an open-source automation software used to configure and deploy systems using Ansible playbooks written in .yaml files.

		We chose Ansible as it provides an easy setup and does not require a new client to be installed on our application servers. Furthermore the configurations is described in yaml thereby not introducing new languages.
Scaling	Docker Swarm	Docker Swarm is a container orchestration tool used for clustering Docker containers. We chose Docker Swarm as the most notable alternative is Kubernetes. Due to the teams limited experience with managing scaled systems, we chose not to use Kubernetes as it is known for being harder to implement than Docker Swarm.
Update strategy	Rolling updates	Rolling updates is an update strategy, that allows a system to update with 0 downtime. Rolling updates is the default update strategy in Docker Swarm and therefore the one we have chosen, as to not make the implementation more complicated than it needs to be.

Table 1: *Tools and dependencies used in minitwit*

1.4 Current state

Table 2 describes the status of the system after the simulator has been shut down.

Total requests	34233
Requests secured	34233
Total unhandled exceptions	18
P99 min response time	94.2 ms
P99 max response time	716 ms
Number of users	116995
Avg followers per user	28.5

Table 2: *System metrics*

Furthermore, the static analysis tool Sonarcloud assesses that the quality of the overall code is good (see appendix B).

2 Process Perspective

2.1 CI/CD Chain

In order to introduce continuous integration and continuous deployment to the program, we used Github Actions, namely workflows. The project included several different workflows like provision and deploy as described in table 3.

Workflow file	Function
AutomaticBuildRelease.Backend	Bumps the patch versioning of the *MinitwitSimulatorAPI* project and creates a release on the github repository.
AutomaticBuildRelease.Frontend	Bumps the patch versioning of the *Minitwit* project and creates a release on the github repository.
WeeklyMinorRelease	Bumps the minor version of the entire program and creates a release each Thursday at 20:00 UTC.
Linters-and_Formatter	Runs Hadolint, Pre-commit, and the dotnet-format linter, when a PR is created, modified, and reopened.
Testing	Runs all tests on the infrastructure, the frontend, and the backend, when a PR is created, modified, and reopened.
Provision-and-Deploy	Provision and deploy swarm and observability servers using Ansible for server provision and Pulumi for infrastructure as code, on pushes to main.
latex-build	Creates a report PDF when changes have been made to the report.

Table 3: *The function of the workflows*

Figure 5 vizualizes the development pipeline all the way from writing code to production. This includes running precommit to check for formatting to passing both UI and integrations tests as well as building and deploying.

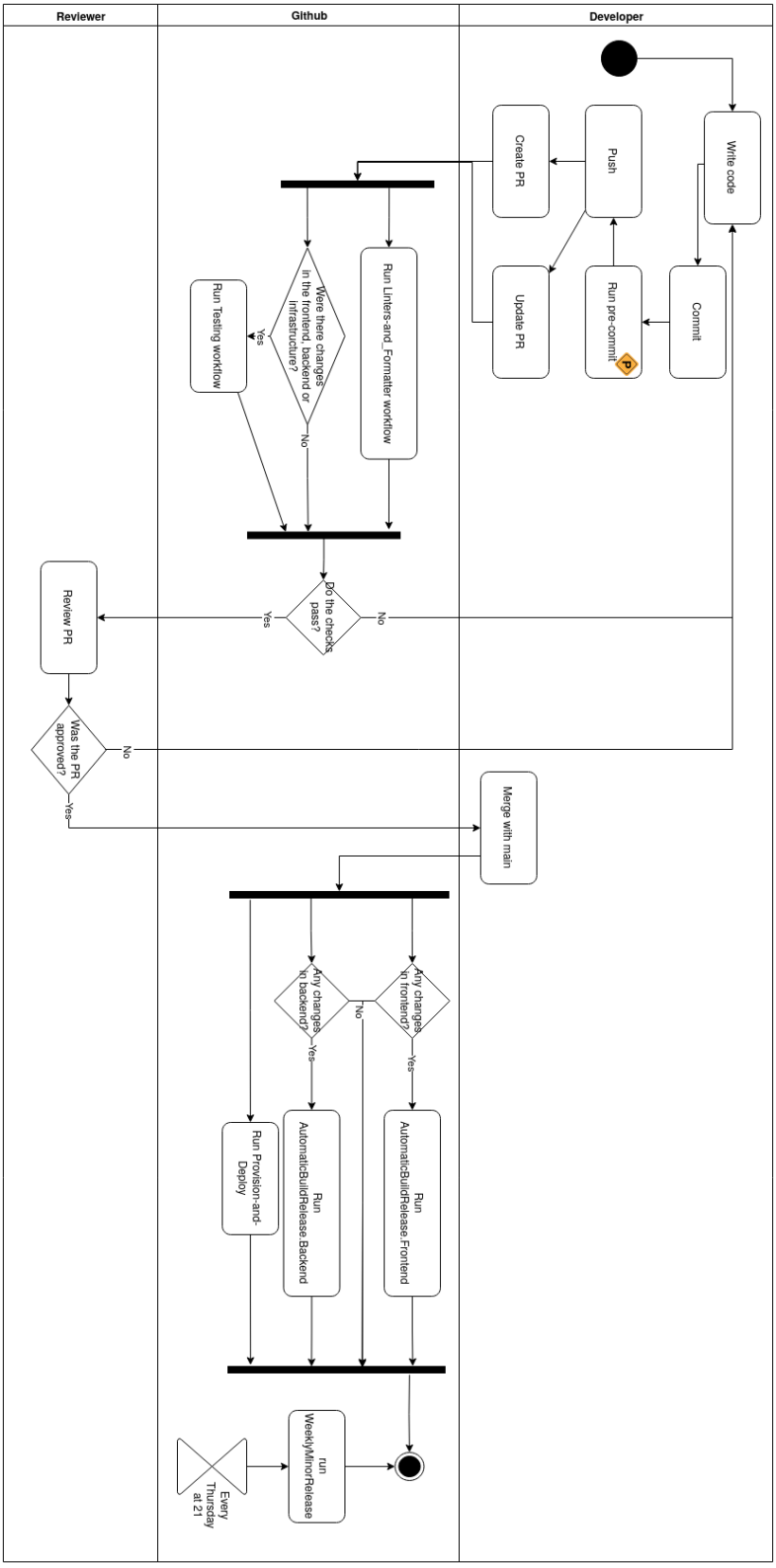

Figure 5: *Activity diagram showcasing the workflows interacting*

2.2 Monitoring and logging

For monitoring we use Grafana. We export metrics by using OpenTelemetry and collect the metrics via Prometheus and lastly push them to Grafana. Our board shows requests duration, errors rate, top 10 unhandled exception endpoints and more (for visualization of the board, see appendix A). We have used the board to get an overview of where to put our focus, ie. which endpoints to improve, which errors to fix, etc. Moreover, we are gaining insights into the health of the system and get an impression of how the backend and frontend handle the requests.

For logging, we use Grafana and Loki. It seemed natural to continue our work with Grafana in order to keep the system setup as simple as possible. The logs are divided into Information, Warning, Debug and Error. All logging statements are placed in the controllers, such that we have information about the users' whereabouts in the system. For example, we log when a user logins whether successful or not.

2.3 Security Assessment

Based on our security assessment (see appendix E), we constructed the following risk matrix:

	Rare	Unlikely	Possible	Likely	Certain
Catastrophic	Exposed Secrets		Depricated Dependencies		Open Ports
Critical	SQL Injection	URL tampering	Password Brute Forcing		
Marginal				Log Injection	
Negligible					
Insignificant					

Figure 6: *Risk matrix from security assessment*

We decided to focus on the scenarios in the red zone, as they would be the cause of most damage. Due to the team not having enough knowledge about how the simulator worked in regards to creating users and logging in, we did not move further with the *Password Brute Forcing* scenario. However, possible solutions could be to incorporate 2FA, password requirements and time-out limitation for an actual system in production.

For *Log Injection*, we made sure to sanitize user inputs, as they previously were put directly into the log, which could be exploited to hide ill-intentioned actions. We thereby hardened the system against injection attacks.

With *Depricated Dependencies*, we chose to integrate the tool *dependabot* into our CI/CD pipeline to watch out for outdated dependencies in our main repository. This helps us ensure that an adversary is not able to take advantage of the team not being aware of vulnerabilities in variuos of the used tools.

Open Ports was deemed the biggest risk, as this was most likely the reason for us being hacked, which we have written about in further details in section 3.1. We used the tool *nmap* to scan the IP address of our application and check for open ports, which showed the following:

Figure 7: *Open ports found by nmap*

We tried to close port 9090, which was used for Prometheus, but it was unclear if we were successful as *nmap* kept showing the port as open, despite seeing the firewall configuration say otherwise on the server itself. This was likely caused by Docker bypassing our firewall, which was handled by the Ansible ufw module, through IP tables.

2.4 Scaling

We use horizontal scaling, as we wanted a more resilient application and limit downtime when deploying. We settled on Docker Swarm over building a custom load balancer or adopting Kubernetes, primarily due to its simplicity and compatibility with the existing system setup. To implement Docker swarm we created a new server in our system and added a manager and a worker node to the swarm.

When attempting to scale by adding another frontend replica, we encountered challenges. In order for the two replicas to work together we needed to handle distributed sessions to remember who is logged in even though the frontend replica is switched. However, we did not succeed in this (we refer to week 11 in the log in appendix D), which is the reason why we only have a single frontend replica.

For the update strategy we went with rolling updates because it is the default update strategy for Docker Swarm.

3 Lessons Learned

3.1 Lesson 1: Getting Hacked

Just a couple of hours after attending the lecture on security, we got hacked, leading us to experience firsthand how important it is to incorporate security in a CI/CD pipeline.

The first suspicion we got was when we discovered, through our monitoring, that the response

time of our server was suddenly very slow. This led us to our Digital Ocean dashboard which showed that the server was using 100% CPU power, which is highly unusual.

From that point the group scoured the server for clues as to what was happening, finding countless calls to `masscan` essentially drowning our server, as well as mysterious installations and what looked like a call to a remote script via a cronjob.

After a few failed attempts to evict the adversary it was decided to destroy the server, as we fortunately had already implemented our Infrastructure as Code, so provisioning and deploying a new server could be done in under half an hour.

After some introspection into our system, we assume that the adversary had gotten access via some open ports that we were unaware were exposed. The ports became exposed in an attempt to make the network function between servers in a docker swarm, which seems to override the firewall.

Learning from this, we have worked to close exposed ports from Docker and finding alternative solutions to setting up the network. Another key takeaway is that because we had the necessary monitoring in place, to figure out that the server was being targeted, as well as having implemented Infrastructure as Code, we were able to detect and react to the attack fairly quickly, giving us only a few hours of downtime.

3.2 Lesson 2: Shift from Vagrant to Ansible-Pulumi

At the beginning of the project, we had chosen to provision our VMs with Vagrant, inspired by the exercises from the course. We realized later that we would have to switch Digital Ocean account at some point due to running out of credit, which then meant we had to streamline the setup of our VMs.

The choice of Configuration Management tool fell upon Ansible, which was supposed to be called by Vagrant in a config server, provisioning the web and monitoring servers. However, it turned out that Vagrant was not the right tool for the job when having more complicated automation and collaboration needs for our project. After many hours of attempting to get Vagrant to work with Ansible, we found out that Vagrant saves local metadata to maintain some state, which was making the provisioning from Ansible and the config server fail[2]. Upon further research we also found out that the creators of Vagrant had created the tool for setting up development enviroments and not for maintaining production infrastructure[3]. Furthermore, Vagrant does not seem to be supported with GitHub Actions, which is likely due to how it handles state.

We decided to cut our losses with Vagrant and search for a more suitable tool that could help us write our infrastructure as code, where the choice fell upon Pulumi. This taught us the importance of thoroughly researching the available tools to navigate through their advantages and drawbacks. It was a valuable lesson to see the difference it made when taking the time to investigate different tools and their properties in order to make an informed decision based on

the knowledge of our system's needs.

3.3 DevOps Style

When reflecting on how we as a group incorporated the style of DevOps into our way of working, we recalled the *Three Ways*, which were the characterising principles for processes and behaviour in DevOps, that consisted of *Flow*, *Feedback* and *Continual Learning and Experimentation*[4].

In regards to the principle of flow, it wasn't hard to adopt the ideas of making our work visible and reducing batch sizes, as all members of the group have previously worked agile in other courses, which also embodies those same ideas. Using a kanban board to track our tasks and their progress not only helped us stay on target, but also helped us with confining each task such that it could be deployed continuously[4].

For the principle of feedback, the concept of peer reviewing via pull requests on GitHub helped install a sense of ownership over the application. Automating the process of not only testing, but also building and deploying the entire application in a continous fashion also allowed for errors to be found and mitigated quickly compared to earlier projects we have worked on, where it was easy for the issues to pile up on each other.

Lastly, with the principle of continual learning and experimentation, having a safe system of work[4] was crucial for us to learn and grow in a secure environment, which in turn allowed for greater experimentation in how to improve the system and its setup. By keeping weekly work logs (see appendix D) and guides easily available, each member would have the same oppertunity to gain a deeper and better understanding of the system as a whole.

4 References

[1] Ansible. *Docker Swarm Module*. 2024. URL: https://docs.ansible.com/ansible/latest/collections/community/docker/docker_swarm_module.html.

[2] DevOps Group e. *Github Issue: Add playbooks in the vagrant provisioning steps*. 2024. URL: https://github.com/devops2024-group-e/itu-minitwit/issues/178.

[3] Hashicorp. *Vagrant vs. Terraform*. 2024. URL: https://developer.hashicorp.com/vagrant/intro/vs/terraform.

[4] Gene Kim. *The DevOPS Handbook - How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. It Revolution Press, 2016. ISBN: 9781942788003.

[5] Microsoft. *Design the infrastructure persistence layer: The Repository Pattern*. 2023. URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design#the-repository-pattern.

[6] OpenTofu. *The OpenTofu Manifesto*. 2024. URL: https://opentofu.org/manifesto/.

[7] OWASP. *OWASP Top Ten*. 2021. URL: https://owasp.org/www-project-top-ten/.

Part

Appendices

Table of Contents

A Grafana

Figure 9: *A snapshot of the Grafana dashboard.*

B Sonarcloud

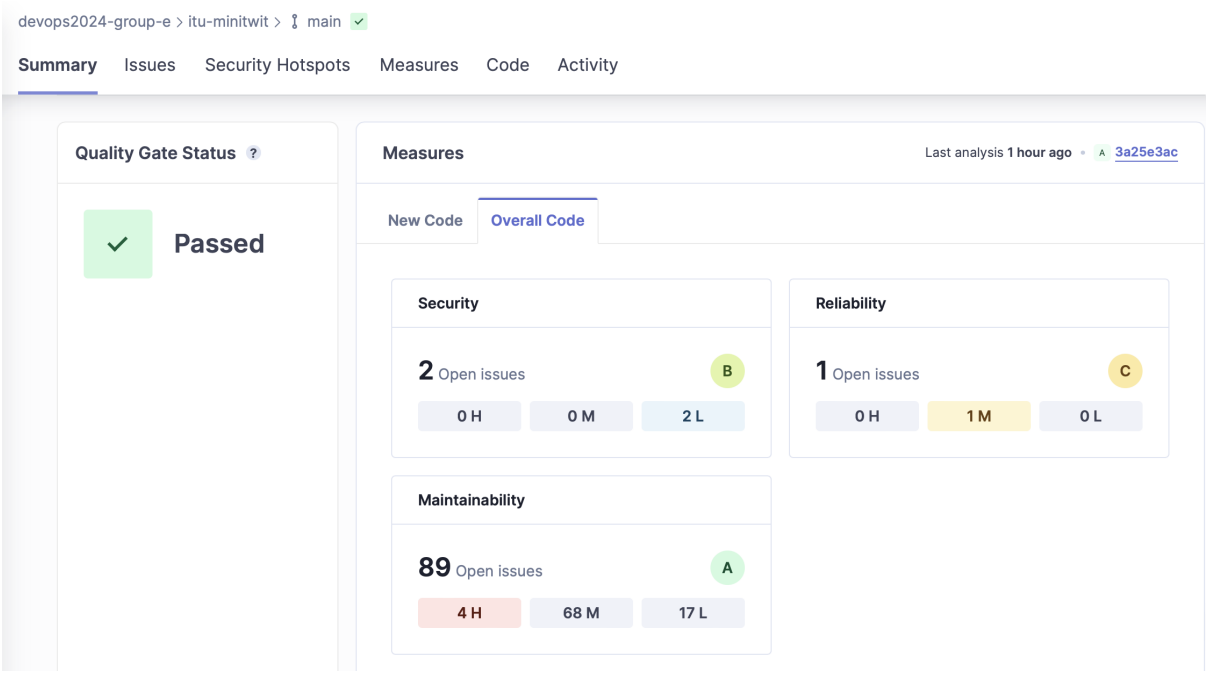

Figure 10: *Sonarcloud code analysis.*

C Decision Log

C.1 Language: C#

We decided to use C# for refactorization. We did this due to C# being object-oriented and a stable language. C# is used by many companies and is therefore known to many. Because C# is used by many there also exists good documentation easily accessible. There exist both micro web-frameworks and more advanced front-end frameworks for C# in case we want to develop the application further. C# is also a part of the .NET platform, which allows for language interoperability and access to the Core framework used for web applications.

C.2 Micro web-framework: Razor Pages

Initially, we decided to work with Scriban as the micro web-framework, as opposed to e.g. Razor Pages, because the refactorization from the original version of Minitwit seemed more direct in Scriban. This is due to the fact that Scriban works with `.html` templates, like we have in the original version, whereas Razor Pages integrates C# in HTML, which would lead to more "merging" of code. Furthermore, Scriban's parser and render are both faster and use less memory than other templates. After trying to find information on Scriban, the group had trouble figuring out how to set it up properly. Comparing the amount of documentation for Scriban versus Razor Pages, we decided to switch the micro web-framework to Razor Pages, as this would allow us to start working with the material faster. Razor Pages also has simple context of structure and is flexible to fit with any application. Another reason for choosing Razor Pages is that each page is self-contained, with the view and code organized together. This makes Razor Pages more organized and easier to work with. With the documentation, flexibility and the organization, Razor Pages felt like a good choice for a micro web-framework.

C.3 Work log: Notion

To start of with, we decided to keep our work log/diary in GitHub, since this automatically provides us with useful information, e.g. dates and authors. Additionally, the work log will follow the code completely, and we can see which code belongs with what log entries. However, we realized after a few weeks that this did not work as intended. The reason was that the action of making a pull request and requesting a review after just a tiny addition to the log seemed too complicated. As a consequence, we ended up keeping our log in Notion and then moving it to GitHub afterwards, which was double the work and definitely not the intention from the start. Therefore, we decided to properly move the log to Notion, which also has the advantage that everyone can follow each other's writing in real time and collaborate in writing simultaneously.

C.4 Database connection: Entity Framework

We have chosen Entity Framework (EF) for our database connection as it works well with .NET. EF has the possibility of generating a model from an already existing database and lets us load data using C# classes. Additionally, EF supports LINQ and works well with SQLite.

C.5 Server host: Digital Ocean

We have chosen Digital Ocean as our cloud hosting service, as it is a cheap cloud hosting service that allows for developers to have quite a lot of control over their server hosting. Furthermore, popular cloud hosting alternatives have a tendency to be more expensive, however typically not by a lot. Even though this increased expense often comes with a lot of additional configuration options, some of the students in the group have limited experience working with cloud servers, wherefore we believed Digital Ocean would be a good choice, as it is free, and there are plenty of guides and good documentation.

C.6 CI/CD: GitHub Actions

For CI/CD we chose GitHub Actions. The reason is that we are already using the GitHub platform and that it is well documented. Furthermore, it is easy to configure secrets and refer to them from the `.yaml` file. Additionally, it collaborates nicely with Digital Ocean which hosts our servers.

C.7 Database: SQLite to PostgreSQL

We chose to use PostgreSQL when we had to start creating our Dockerfile. We looked for a Microsoft or another official Docker image from a good source (that is from a big, known company, that we can trust). But we could not find one and we saw that Helge used something he created himself. Thus, we had to choose among some of the databases that has been containerized. That were the first two requirements. The third requirement is that we need a relational database. This excluded well-known document databases like MongoDB. At the end we had two good options. Microsoft SQL and PostgreSQL.

Feature-wise they offer the same things. They are both good at handling concurrent requests and have transactions. They both offer the possibility to add indexes, and to have multiple servers act as a cluster in order to handle even more requests.

We chose the solution that we have had experience with before. Both in our previous course *Introduction to Database Systems*, and in previous projects done at ITU. We have had an abundance of new technologies throughout our times as students, so it felt almost relieving to not use a new one again.

C.8 Monitoring: Grafana

Looking for a tool to visualize the metrics of dotnet applications, and maybe other tools in our tech stack as well, required a versatile tool. That is, we needed to have a tool where we could search and visualize the current status of our application. Obviously, a lot of tools can handle this. For that reason it is nice to see that the tool OpenTelemetry is announced and widely used - both as a collector and exporter.

Thus, a requirement is that the monitoring system has to be able to integrate with OpenTelemetry in order to make it easier to change monitoring system if required.

Furthermore, it would be prefered to have an open source and free of charge product that is being maintained.

C.9 Metrics collector: Prometheus

There has not been an elaborate thought process about the choice of Prometheus. It is open source and can collaborate with the OpenTelemetry exporter. Additionally, it is able to connect to the Docker metrics as well.

C.10 Database migration: Digital Ocean

We have considered hosting a separate server on Digital Ocean, and using Digital Ocean's own database service. We considered Digital Ocean's options, due to the wide array of databases that they support. However, when researching about troubleshooting, and problems using this solution, we found very few answers from crowd sourced platforms such as StackOverflow, and a lot of guides from the Digital Ocean docs themselves.

C.11 Logging stack: Grafana + Loki + Promtail

We believe that the amount of applications used can get very bloated. Therefore, we have decided to use Grafana-Loki, as we have already setup Grafana last week when implementing monitoring.

C.12 Quality of code analysis: SonarCloud

SonarCloud is a cloud based code analysis service, which can be integrated into a GitHub repository, in order to analyse the code in said repository. We added this software as it was an exercise in the course.

C.13 Linters: Pre-commit + Hadolint + Dotnet-Format

Pre-Commit is a framework for managing pre-commit hooks. This can be used for automatically fixing code format and removing debug statements.We chose this tool as multiple actions can be integrated into one hook. As we were tasked with integrating 3 linters into the system.

Hadolint is an open-source Dockerfile linter, that enforces best practices when building docker image. Due to the nature of the server setup, we have quite a few Dockerfiles. Therefore, we found it fitting to have a Dockerfile linter. From the alternative presented (Dockerlinter) Hadolint seems to be the more popular one as it is haskell based rather than node.js based.

Dotnet-Format is a formatting tool, which is included in the .NET 6 SDK and onwards. It applies style preferences to a project which can configured in an .editorconfig file.

C.14 Configuration management: Ansible

We realise now that we need a tool that easily manages the desired state of our running VMs. Especially now that we are going to create more VMs and that we have to move our infrastructure before the 19th of April to another Digital Ocean account.

For that reason we need a simple way to have a configuration of our VMs defined that manages our provisioning. Our requirements are the following:

- Simple setup
- Should be pushable i.e. as soon as we have changed the configuration, then it should push the desired state out to our servers
- Keeps track if specific portions of the provisioning scripts already have been run
- Does not require us to learn a new language

We have been looking at different players in the market. Especially we have been looking at the CNCF and asking colleagues. Here are some of the products we have looked at:

- Ansible
- Fabric - Not really a configuration management system but a “simple” push-based script system
- Chef

C.14.1 Fabric

Fabric seemed nice, however it lacked some of the basic things that we wanted to solve, such as the idempotence of the desired state of the VM. This we would have to script ourselves. Also

we would have to add another language to our repo, that is Python, which would require us to setup our development container again to have both dotnet and Python. Our experience with that was that it was very hard to get right, and we finally have a setup now that works on all of our laptops. So we did not want to break that.

C.14.2 Chef

Chef is a configuration management system, just like Ansible as well. It provides a hybrid solution of pull- and push-based. That is, we push the configuration management changes to a central server and then the VMs will pull the changes from the server (probably after some interval).

As can be seen this also requires that the VMs has the Chef Client installed and configured to point at the Chef Server. This provides the nice feature that the Chef Server do not have to know which servers to configure. It is up to the Chef Clients to know where to pull the configurations from.

The downside of this setup is the complexity. It seems like a lot of things that needs to be taken care of, and the fact that we would have a load overhead by adding clients to our small web server VMs would maybe

In order to create a configuration we would have to create what they call cookbooks. This is written in Ruby as far as we can see. The nice thing about this is that there are testing frameworks implemented, which provides a way to be sure that the configuration works and configures as we intend it. However, while experimenting with Vagrant it has become obvious that our group does not have sufficient knowledge in Ruby and we would currently rather invest time in becoming better at using the observability tools that we have, rather than learning a new language.

From a security point of view Chef uses some of the protocols that we have already opened ports for. That is, it uses standard HTTP protocol to communicate between the server and nodes, and authorizes through certificates.

C.14.3 Ansible

Some of the nice features in Ansible is that the setup seems fairly simple:

- One server to distribute to a list of servers
- No additional clients to install
- No code for defining configurations, it uses a YAML to define the configuration of the VMs

The downside is that there is a central server that needs to have a list of the created VMs, which then needs to be maintained. Furthermore, from a security perspective, we would have

to allow ssh and that the Ansible server would have to store the private ssh key to access all of the servers. We have a hard time trying to understand how we are supposed to push this to the Ansible server and then run it? Would that be another ssh session?

Looking a bit further into it is seems like the way to automate the configurations of the server can be done with a proprietary product called Ansible Automation, which requires a license. But some has been able to create a custom GitHub Action that can push these changes to the servers. This requires that the ssh port is open to the public, which it already is.

As with Chef, Ansible also has their own term for a configuration, i.e. a playbook. The term is from American sports, where each task in the playbook is a play that constitutes the playbook. Each play in the playbook is idempotent. Thus, it seems to provide simple idempotence, which is fine in most cases, but it some cases it may require some more advanced idempotence.

C.14.4 Why do we choose Ansible?

We chose Ansible because the setup seems simple. It provides the features that we need i.e. an easy setup that does not require a new client to be installed on our application servers. Configurations described in YAML and not in some other language like Ruby or Python.

C.15 Adding E2E and integration tests early

Early in the project process we were given some integrations and end-to-end (E2E) tests for the frontend and Simulator API, respectively. This was beneficial when we had to develop our system to begin with, but we quickly realised that in order to be comfortable, and to ensure that our future changes in the software did not break our interface with the simulator, then we had to add them to our CI pipeline via the GitHub Actions workflows.

However, we realised again that having them in Python would make it harder for us to extend our tests or create more tests as we progress further in our maintenance cycle, wherefore we needed to refactor it to C#.

C.16 Infrastructure as Code (IaC): Pulumi

We have lately been a bit frustrated at Vagrant. It seems like it lacks some collaborative features that makes it hard to work with. Furthermore, trying to make it collaborate with a configuration management tool, like Ansible, proved to be very challenging and time consuming considering that it should “just be plug and play”. We did not make that work, so to eliminate a pain that we currently have, we are looking towards encoding our infrastructure as code now. What is our requirements?

- Should be easy to work with

  - That is, we would prefer to keep the configuration in **YAML** or **C#** as we do not want to increase complexity of learning a new language again

- Can use it in a **GitHub Actions workflow** such that we can create changes in the Minitwit repository, create a pull-request, and then push changes in production when everything is working
- This is not a requirement, but it would be nice if we can use an **open source** tool.
- A tool which can work for multiple cloud providers, such that we can change provider if we want to. However, since we are at Digital Ocean (which we are pretty happy with so far) then we want them to support Digital Ocean.

We looked at the following options:

- Vagrant - Is it still a good option or not?
- Terraform - As stated in a later lecture is something that we are going to look at
- Pulumi - A tool we stumbled upon in a Digital Ocean video
- OpenTofu
- Write code ourselves with a custom client to Digital Ocean

C.16.1 Vagrant

Is a very nice tool for creating infrastructure quickly. What we have experienced so far, however, is that it is hard to collaborate within. It saves metadata files on the local machine to keep some sort of state. It only pushes the creator's `ssh` key to the server, and not the other's `ssh` key, making us copy paste `ssh` keys to the server manually. Last but not least, it is written in a language that none of us have experience with (Ruby). There is not any GitHub Actions for using Vagrant, which we think has something to do with the way that it handles its state. Looking at Vagrant's comparison between Vagrant and Terraform (which is also a tool made by hashicorp) the following is stated:

„Vagrant is a tool for managing development environments and Terraform is a tool for building infrastructure... and more features provided by Vagrant to ease development environment usage... Vagrant is for development environments.[3]“

By this description, Hashicorp has not intended to build Vagrant for the scenario that we use it for today. That is, they do not and will not improve the tool in the direction that we intend to use it. Furthermore, it does not make sense for us to keep Vagrant to setup a local development environment because we already have Docker and Docker Compose that takes care of that.

C.16.2 Terraform

As we can see in the above section, Terraform is a product that is used for the specific functionality that we need. It can integrate with our GitHub workflow. It does so with Terraform Cloud to keep the state of the infrastructure. It can create infrastructure on multiple cloud vendors. It also has support for Digital Ocean.

One of the downsides is that they introduce their own language: HCL. This is something that we do not want to have. We want to limit the amount of languages that our repository contains to a minimum. Although we can see that they actually have what they call a CDK (Cloud Development Kit) for Terraform, however, we doubt that is their first priority to maintain. An upside to the CDK is that it can add testing to Terraform.

Terraform is not open source, which makes it harder to trust the tool. A quick search on the internet yields that others have forked a former repository of Terraform and created a Terraform look-a-like, called OpenTofu, which reveals that the community does not trust Hashicorp's development of the tool, now that they have transitioned from open source to a Business Source License.

C.16.3 Pulumi

Pulumi is similar to Terraform, in the way that it also provides a cloud solution to keep state of the infrastructure. It does not offer a new language to write the infrastructure, but instead offers SDKs in languages that we already know. Both Pulumi and Terraform follow what is called Desired State Infrastructure, meaning that they try to keep the infrastructure in the desired state that the code describes.

It contains support for CI / CD tools, so we can integrate it into our existing workflows. In contrast to Terraform, it is open source. Pulumi seems to be widely adopted and is known for being easy for developers to get started.

C.16.4 OpenTofu

OpenTofu is a forked version of Terraform. It keeps the structure in HCL. What makes this interesting is that the community has actively taken a decision to maintain this project in order for it to not be a BSL tool, distancing them from Hashicorp's decision with Terraform. This can be seen in the OpenTofu Manifesto[6]. Another advantage to OpenTofu is that it has later been adopted in the Linux Foundation and for sure is here to stay.

As it is a copy of Terraform it has some of the same features and downsides. This can be seen by having their own custom language to describe the infrastructure called OpenTofu Language. OpenTofu has not done the same thing as Terraform in terms of creating a CDK. That means the only option is the OpenTofu Language.

Another downside is the fact that we are unsure whether there is support for Digital Ocean as a provider. We would think that there would be, but it is not clear and we will need to spend time looking into it.

C.16.5 Write C# code with Digital Ocean client

This option may not be the best option. Compared to the others we will have a tight coupling to Digital Ocean. This will make it challenging for us to change provider in the future, if that is needed. This will also require us to keep an eye out for the changes to the general API and make adjustments to the custom client that we would have to make. We could also find a client that is made for us, written in C#, but then we would have to find one that is being maintained.

We have realised that the maintenance of just written code for the infrastructure, becomes a project in itself. We would have to manage an abundance of other dependencies, which is taken care of from the other projects. Also we do not have any testing capabilities as the other tools have.

Furthermore, we will not benefit from things that others may have created. A large community for an open source tool allows for sharing solutions and code for that specific tool. We do not really have that option here, as we are going to create a complete new tool with this option.

C.17 Scaling: Docker Swarm

We need to add horizontal scaling to our system.

Our requirements

- Open source
- Works well with Ansible/Docker/Pulumi.

C.17.1 Docker Swarm

[1]

- Open source
- It is based on the Docker API, and therefore is **lightweight**, and **easy to use**.
- Docker Swarm allows for limited scaling, and is not recommended for big systems, so if Minitwit ever becomes a bigger service, this could potentially be a problem.
- It is easy to add new nodes to existing clusters as both manager and worker
- Compared to other tools like kubernetes

- Simple to install compared to Kubernetes
- Same common language that we are already using to navigate in the structure
- Limited guides on interactions between Docker Swarm and Ansible.

C.17.2 Building our own redundant load balancer

- Time-consuming to create
- Less information on how to build it
- Probably more challenging to add new nodes than for Docker Swarm

C.17.3 Kubernetes

- Takes more planning to implement than Docker Swarm
- Has a quite steep learning curve
- It can sustain and manage large architectures and complex workloads
- Has a GUI

C.18 Upgrade/Update Strategy: Rolling updates

Rolling updates is the default update strategy in Docker Swarm and therefore the one we have chosen so as to not make the implementation more complicated and "home-made" than it needs to be.

C.19 Precommit

D Log

D.1 Week 1 - 02/02/24

Task: Our task was to make the `minitwit.py` run with python v3 instead of python v2.

Steps taken:

1. We copied the source code from the server

```
scp -r student@164.90.160.52:/home/student/itu-minitwit ~/Desktop/
```

1. We also copied the database which was saved as a `*.db` file with

```
$ scp student@164.90.160.52:/tmp/minitwit.db \textasciitilde{}/Desktop/itu{-}minitwit
```

3. Then we created a Github organization with a repository called itu-minitwit
4. We clone the minitwit repo

```
$ git clone https://github.com/devops2024-group-e/itu-minitwit.git
```

5. We copied the files from `~/Desktop` to the location where we cloned the repo
6. We tried to run the application by using the `control.sh` script. But got an error message that it were missing the database. We saw that the database file actually should be located in the `/tmp` and not in the current working directory of the application. So we placed the database in `/tmp` and then it worked.
7. We then started doing the steps to convert the application from python 2 to python 3. First we used the `2to3` tool to convert to see what has to be changed.

- We changed the reading of the database file to read it as `UTF-8`
- Added parenthesis to the print statement
- In the tests we changed the assertions to use `rv.text` instead of `rv.data` as the `rv.data` is in a binary format and that we are trying to look for a particular string.

8. After that we ran the `shellcheck` on `control.sh`. Here we added the shebang (`#!/bin/bash`) to the top of the file.
9. After all of this we tried to look into the difference between `python` and `python3` command (take a look here). After looking into it we decided to just use the `python3` command in order to make sure that the tools we use actually runs python3.

D.2 Week 2 - 09/02/24

Task: Our task was to refactor minitwit to C# and RazorPages.

Steps taken:

D.2.1 Refactorization:

index.cshtml:

- Made @page {tl : string} optional

index.cshtml.cs:

- Added a string path

index.cshtml:

- Display tweets
- for loop for all messages

index.cshtml.cs:

- Created public list for messages
- We copied the original .css file and added in the new .css
- Checking if there is messages and printing them
- We tried to install codegenerator - it didn't work. Line of command:
- We created login.cshtml and linked it to page and model LoginModel

Layout:

1. We created the 'Shared' folder under 'Pages' to hold UI that we wish to use across different pages. Here we then added `_Layout.cshtml` and rewrote the code in `layout.html` from the python source code to match the correct format of '.cshtml' files to encompass the same UI format.
2. We added the `style.css` file from the original project and added it to the 'wwwroot' folder under the name `main.css`. This was done to connect the html and css properly, so that the refactored version of the website now matched the original one in terms of looks.

Login:

1. Created a `Login.cshtml` that contains the same logic as in the python version by adding pieces of the page one at a time.
2. Added an in memory user (let us call it a dummy user) as we do not have the database connected for now and we just want to see if the UI logic actually works when a user tries to login
3. Adding a session that should store the `userid` as it does in the python application. We do this by adding the following lines as settings in the `Program.cs`:

```
builder.Services.AddDistributedMemoryCache();

builder.Services.AddSession(options =>
{
    options.IdleTimeout = TimeSpan.FromMinutes(5);
    options.Cookie.HttpOnly = true;
    options.Cookie.IsEssential = true;
});


app.UseSession();
```

4. Now we can get and set session data by using `this.HttpContext.Session.SetString()` and `this.HttpContext.Session.GetString()` in a `PageModel` class
5. Add if statement that if the login is successful then it should store `userid` and redirect to `/Timeline` (Timeline should take care if it should show a public or user timeline)

Register:

1. Created the `Register.cshtml` and added the same UI logic that is present in the corresponding python template `register.html`
2. Then the `Register.cshtml.cs` page model that handles the request to the site was created, with the model that is necessary in order to register
3. We tested the site if it worked as it should but figured that the page model did not capture what was typed because we missed the `BindProperty` on the properties in the page model. So we added that
4. After, we had to adjust the validation if statement to work correctly due to a missing negation of the `Email.Contain("@")`

Timeline:

- Trying to add switches between the different timelines. Trying to make it show different titles based on user.
  - I cannot make it work by "just" adding another if/else in time index.cs file.
  - I don't where the title "Public" on public timeline gets set as it says "Public Timeline" in the switch in index.cs file.
  - Turns out the connections we had made before relied on getting the path and doing some operations on that, but that is the wrong way to get the path. You just get the path as input to the `OnGet` method, and from there I make a simple switch and send it on to the html.
  - A lot of help came from: `https://www.learnrazorpages.com/`
    - `https://www.learnrazorpages.com/razor-pages/routing`
    - `https://www.learnrazorpages.com/razor-pages/forms`
- Added a bit of different views for logged-in user, other-user and public timeline.
- Added shared view for submitting a message.
- A lot of functionality still depends of having a database where users and messages are stored
  - Knowing whether we have a logged-in user
  - Knowing the connection between users (following/not following)
  - Having the messages stored in a database and store more messages as times goes on

D.2.2 Connecting to a database:

1. In Setup.sh we added the dotnet ef tool version 7.0.1
2. Added designpacket ****`dotnet add package [Microsoft.EntityFrameworkCore.Design](http://Micro version 7.0.1`
3. Added sqllite package to the project `dotnet add package [Microsoft.EntityFrameworkCore.](http://Mi 7.0.1`
4. Referenced database, `dotnet ef dbcontext scaffold "Datasource=/temp/minitwit.db" Microsoft.EntityFrameworkCore.Sqlite -o Models context-dir .`
5. Scaffold wants us to provide a connection string (first argument) and which sql provider we use (second argument)
6. We want to place the models and context a specific place, which is specified at the end of the command
7. Injected dependencies - adding services to container

D.2.3 Issues with creating endpoints

Since we tried to use the "pure" razor template we had some trouble trying to create the exact same endpoints that was present in the `[minitwit.py](http://minitwit.py)` and `minitwit_tests.py`. We tried to create our own custom routing via:

```
services.AddRazorPagesOptions(options => {
    options.Conventions.AddPageRoute("<Ref to page>", "new/route/to/page");
});
```

However, the above solution to our endpoint path problems couldn't mitigate the issue. For that reason we looked at what we could do in order to solve our issue. And in the official documentation we found another way to create Razor pages with the MVC architecture. So we decided to pivot and create an MVC project. Luckily we could copy the contents of both `*.cshtml` pages and `*.cshtml.cs` files from one project to another.

D.3 Week 3 - 16/02/24

Task: Our task was to create a simulation API and deploy the program

Steps taken:

D.3.1 Setting up a basic VM in Digital Ocean using Vagrant

1. We installed Vagrant on our machines (mac: `brew install --cask vagrant`)
2. Followed this guide in order to generate keypairs `https://www.digitalocean.com/community/tutorials/how-to-set-up-ssh-keys-on-ubuntu-1804`
3. Registered public key on digital ocean and received a digital ocean token.
4. We set two environment variables in "which file?"
5. We run this command `$ vagrant plugin install vagrant-digitalocean`
6. We then run `vagrant up` with the exercise files and got it to work
7. We made sure to destroy the dbserver and webserver.
8. We then made a new branch in our project and copied the vagrantfile in the exercises session to our own project.
9. We renamed dbserver and webserver to dbserver_new and webserver_new in order to being able to differ them from the exercise servers (important that we destroyed the right ones)

10. We noticed that the IP address for the dbserver that was needed in order to get the webserver didn't work now
11. We then figured out that we had to name them with dash instead in order for the names to work with the vagrantfile, now we renamed them to dbserver-new and webserver-new
12. We started making the vagrantfile more personalized to our program and included what we need. Like dotnet, sqlite etc.

D.3.2 Setting up a simulation API

1. We followed this tutorial: `https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-7.0&tabs=visual-studio-code` up until the point about the controller being scaffolded. There we deviated and used the same bash script that we used for the original system. This helped us set up an API skeleton with connected Swagger for us to have a starting point to work with.
2. We copied over the *Controllers*, *Models* and *ViewModels* folders in order to have the same logic as the original refactored Minitwit system.
3. We rewrote the different controllers to then fit the logic of the python API simulator given by Helge. This mostly entailed returning the specified status codes instead of redirecting to certain pages, since the API isn't concerned with the view of the system.
   (a) We checked that the controllers worked as expected by inspecting swagger and testing the HTTP requests with different input to see if the methods behaved accordingly.
4. After finishing writing the API, we added a folder *Test*, which contains Helge's python API simulator `[minitwit_sim_api.py](https://github.com/itu-devops/lecture_notes/blob/master/sess` and ran it against our API. The same folder contains guides on how to run these programs against each other.
   - Here we encountered consistently error codes in the form of 400's and 500's. After troubleshooting and inspecting the files given by Helge, we figured out that our minitwit project were using the session ID in order to check whether or not a user was logged in. This clashed with the API tests which were checking whether the "Authorization" part of the HTTP header was equal to a certain code. We then rewrote the login function for our API along with the functions affected by no longer being able to pull information from the session ID.
5. After rewriting the API, all test but 1 from Helge's API simulator passed and the work was added to that weeks release.

Possible issues if recreating:

1. Note that when running the `scaffold-db.sh`, it may transform the model classes to use `byte[]` types instead of the original `string`.

D.3.3 Managing the database through the dotnet App

In the process of refactoring from python to C#, we still had some minor things that needed to be moved when we look specifically at the database. These things are:

- Create the database (This removes the last dependency to flask in the `[minitwit.py](http://minitwit.py`
app)
- Get the connectionstring to the database from the configuration
- Add Composite Primary Key to Follwers table such that the Entity Framework can track changes to it

We did the changes by in the following order:

1. First we looked at the website in order to see how we can create the db. We figured that it was possible to do it with `dotnet ef database update` at the respective project where the Entity Framework `MinitwitContext.cs` file is.
2. We then the command line `dotnet ef database update` to the `[control.sh](http://control.sh)` file. But i ran into an issue that i changed the working directory.
   (a) This was fixed by looking into how to change the working directory and executing a particular command at that location.
3. Then we changed the `Makefile` at init to use the `[control.sh](http://control.sh) init`.
4. Now that we can create the database with dotnet, we removed the `[minitwit.py](http://minitwit.py)`
5. After that we looked into how we can get the connectionstring from configuration. We found out that .NET Framework has created some different providers that we can use. Since there already was two settings files, namely `appsettings.json` and `appsettings.Development.json` we decided to use them, and incorporate the possibility to override them using the Environment Variables with a `Minitwit_*` prefix. So we included this in the `program.cs`.

```
// Register source of configurations
builder.Configuration
    .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
    .AddJsonFile($"appsettings.{builder.Environment.EnvironmentName}.json", optiona
    .AddEnvironmentVariables(prefix: "Minitwit_");
```

6. Then we added the connection-string in the settings files.

7. And included the connection-string at startup of Entity Framework in the application with

```
builder.Services.AddDbContext<MinitwitContext>(options =>
{
    options.UseSqlite(builder.Configuration.GetConnectionString("MinitwitDatabase")
});
```

8. Last but not least. To finish the changes with the database we changed the database schema to have a composite primary key. Usually this would be a good idea to do in a SQL Schema. But since the database is being initialized by EntityFramework, then this description was changed in the `MinitwitContext.cs` file. To be more specific we added a line with `HasKey(e => new { e.WhoId, e.WhomId });` to the Follower entity.

9. But this change has to be reflected on the `Follower.cs` model as well. So we had to remove the possibility that the `WhoId` and `WhomId` could be `null`

D.4 Week 4 - 23/02/24

Task: Finish creating an API for the simulator to call. Setup CI/CD. Deploy Minitwit

Steps taken:

D.4.1 API:

1. We realized that the API was missing an endpoint `/latest`. We wrote a separate controller to handle this functionality, where it finds the largest command ID under the table 'Latest' in the database and returns it.

2. We also missed that every HTTP request called a method called `UpdateLatest` as the first order of business. We rewrote this function to C# where it adds the current command ID to the data base under the table 'Latest'.

D.4.2 CI/CD:

1. We copy Helge's example yaml file and work from this, by renaming and restructuring to fit our own setup.

2. We stumble upon the cache-to and cache-from options but fail to understand what these actually do and how it works. We decide to delete them from our yaml file for now, but return later or ask for help.

    - By looking around online, we stumbled across this link: `https://docs.docker.com/build/cache/backends/gha/` where it seemed that it wasn't necessary to manually insert the URL and token, since it is given earlier in the file. So we kept the formatting as seen in the link, which also was present in Helge's original file

3. Because we have our images in the GitHub container repository, we changed the login from docker hub to ghcr(github container registry)
    - The docker images were moved to here in order to keep information such as password to the servers private
    - We used this link: `https://github.com/docker/login-action` and modified what was written under "Service account based authentication" to fit to ghcr
4. We removed the part about testing, since we currently don't have a docker image for testing our code. This should be worked in at a later date.

Task: Ensure that the backend (API), the frontend, and the database are running on the server

Steps taken:

D.4.3 Vagrant

1. We realised that the `Vagrantfile` set up last week, was in large part not usable for the purposes that we needed, since all we really needed was to set up the server and accessing it, we rewrote the `Vagrantfile`.
2. We wrote a guide on how to create and access the server using vagrant: Connecting to the Digital Ocean server using vagrant
3. Later this approach was changed by adding the following to the `.ssh/config` file.

```
host devops
    Hostname 161.35.78.128
    User root
    Identityfile ~/.ssh/do_ssh_key
```

   Which meant the server could be reached by simply typing `ssh devops`.

D.4.4 Docker

1. We created dockerfiles and docker hub repositories for the frontend, the backend, and the database.
2. We built these images locally and then pushed them to pinkvinus repo on dockerhub.
3. However, we realised that we didn't like to have the images publicly available to we created our docker images with `ghcr.io/devops2024-group-e/<image name>:<version tag>`. We need the first part of the docker image name tag because it references to the registry where it should push and later get the docker image from. So an example of a command in the shell we used is:

```
docker build -f Dockerfile.backend . -t ghcr.io/devops2024-group-e/backend.minitwit
```

4. The way we published the images to see if it worked was with the following commands in order:

```
docker build -f Dockerfile.backend . -t ghcr.io/devops2024-group-e/backend.minitwit:lat
docker build -f Dockerfile.frontend . -t ghcr.io/devops2024-group-e/frontend.minitwit:l
docker build -f Dockerfile.database . -t ghcr.io/devops2024-group-e/db.minitwit:latest

# Authenticate to the github private container registry
docker login ghcr.io -u <github username> -p <github token>

# Push the build images to the container registry
docker push ghcr.io/devops2024-group-e/backend.minitwit:latest
docker push ghcr.io/devops2024-group-e/frontend.minitwit:latest
docker push ghcr.io/devops2024-group-e/db.minitwit:latest
```

5. We ran the docker images locally first and it worked perfectly. We could connect to the database and everything. We tested this by performing:

```
docker run -p 5432:5432 --name database -d ghcr.io/devops2024-group-e/db.minitwit:lates
docker run -p 80:80 --name frontend -d ghcr.io/devops2024-group-e/frontend.minitwit:lat
docker run -p 8080:8080 --name backend -d ghcr.io/devops2024-group-e/backend.minitwit:l
```

   And then we tried to poke around in the website to see if it actually had access to the database. We also used the `minitwit_test.py` to test that the site still worked accordingly.

6. Now we add a `remote-server` directory that contains the scripts that deploys and setups the server. In there we create `[init.sh](http://init.sh)` which create the docker containers when the server is being created and we create `[deploy.sh](http://deploy.sh)` which is used when a new image of backend and frontend has been created.

7. We then used vagrant up to see if it worked. It created the server... but. It cannot create and use the containers based on the docker images we have created because it has been built by the platform `linux/arm64` but the server needs `linux/amd64`.

   (a) To resolve this we have looked at using the —platform flag. This did not solve it. Actually we had a hard time trying to build the images on Mac

   (b) We then built the images from the workflow and this made it work

8. On the server, we removed the created containers that could not run and ran `[init.sh](http://init.sh)` in the `/minitwit` directory. And it almost worked. It could not connect to the database.

   (a) No matter what if we change the connectionstring to use `Host=localhost` or `Host=127.0.0.1` with port `5432` it does not work.

(b) We decided to create a docker compose file and express the network. Here we can reference the name of the database container in the connectionstring in the `Host` part of the connectionstring.

i. IT WORKED!!!

9. We create the docker compose file in the `remote-server` directory such that when we provision the server then it is included.

(a) Also modifying the `[deploy.sh](http://deploy.sh)` script to:

```
echo $DOCKER_PASSWORD | docker login ghcr.io -u $DOCKER_USERNAME --password-stdin
docker run --platform linux/amd64 -p 80:80 -d ghcr.io/devops2024-group-e/frontend.m
docker run --platform linux/amd64 -p 8080:8080 -d ghcr.io/devops2024-group-e/backer
```

D.5 Week 5 - 01/03/24

Task: Clean repository. Abstract database. Include testing in our workflow.

Steps taken:

D.5.1 Clean repository:

D.5.2 Abstract database:

- In order to abstract the database layer, we have created repository classes in a separate Infrastructure project which can be called from the application and the API. We have also added tests for these in a separate test project. The benefit of this is that we don't write SQL directly in the methods, but instead we abstract them out such that they can be reused.

1. Added Repository + Repository interface
2. Replaced calls to `MinitwitContext` with calls to the individual repositories (with dependency injection)
3. Made sure in `Program.cs` that the correct repositories are given in the dependency injection when building
4. Wrote tests for the repositories with mock database
    - `dotnet add package Microsoft.EntityFrameworkCore.InMemory --version 7.0.2` package added to be able to test the database abstraction on an in-memory mock database.

D.5.3 Add testing to workflow:

We started by creating a "skeleton" workflow that would only run when a pull request was opened, which consisted of setting up both python and Dotnet along with their needed dependencies to then test that the correct versions had been chosen.

We then removed the python since we only used it to test if the workflow worked as expected. We then set up the database and ran the frontend in order to run the endpoint tests, which was the test setup we at that time could incorporate into the workflow.

Test for other parts of the system will be added at a later stage, when they are available.

We then played around with the best option for which types of pull requests should trigger this workflow. After several tries, we ended up with having no restrictions on the type, as being too specific caused the workflow not to be activated when it needed to be. This means that closing a PR and opening it again reruns the workflow whether it passed or not the first time and likewise for other situations. This ensured that we always tested the new code before merging it into main.

D.5.4 Fix last step of workflow: deploy step

The workflow prints the following in the deploy step:

```
Error: Cannot perform an interactive login from a non TTY device
no configuration file provided: not found
no configuration file provided: not found
no configuration file provided: not found
```

1. Found that the issue could be related to not providing credentials to the `password-stdin` flag of `docker login`.
2. Create `[test.sh](http://test.sh)` script on the server with the original `docker login` line of the script and set file permission with `chmod +x /minitwit/test.sh`. Ran `ssh <user>@<server> -i <ssh key> -o StrictHostKeyChecking=no '/minitwit/test.sh'` and can confirm that same output
3. Verified that the `DOCKER_USERNAME` and `DOCKER_PASSWORD` was on the server
4. Change the login line to be `docker login ghcr.io -u "${DOCKER_USERNAME}" -p "${DOCKER_PASSWORD}"`
5. Tried the test script again. Now it works.
6. Remove `[test.sh](http://test.sh)` on the server
7. Add the new `docker login` line in `[init.sh](http://init.sh)` and `[deploy.sh](http://deploy.sh)` script in our repo
8. After PR approval and merge, add script changes to the server manually

D.6 Week 6 - 08/03/24

Tasks: Add monitoring. Fix unresolved issues from last week. Refine logging.

Steps taken:

D.6.1 Beginning Migrate Database

1. Since we do not have a configuration management tool yet we decided to create the database as a managed service. We need to setup Ansible so that we have an automated way to setup the database in the future. An issue is created.
2. In the DigitalOcean UI we only allowed private services to connect and Andreas public IP
3. Now we are installing PGAdmin so that we can connect.
4. In order to connect to database securely we do the following:
   (a) Go into DigitalOcean
   (b) Navigate to the database resource
   (c) Download CA certificate
      i. Install certificate in system
   (d) Take login parameters and insert into connection details in PGAdmin when adding a connection(Use the public connection not PVC)
5. Create minitwit database
6. Create minitwit-web user
7. Execute `schema.sql` on minitwit database
8. Create role and add permissions to select, delete, update, and insert records in the public schema on minitwit database

```
CREATE ROLE minitwit_app_executer;
GRANT SELECT, UPDATE, INSERT, DELETE ON ALL TABLES IN SCHEMA public TO minitwit_app
```

9. We(or I, Andreas) got tired and decided to not perform the migration until we have more time...

D.6.2 Add monitoring:

1. We first added a new configuration similar to our minitwit configuration in our vagrant file

2. Move our `remote-server` directory into an `.infrastructure` directory and changed the synced folder in our vagrant file
3. Change the synced directory on our `minitwit-monitor` vagrant config
4. Add a docker compose file containing the grafana application and expose on port 80
5. Login with admin and changed password
6. Change the docker compose file to run prometheus
   (a) Add `prometheus.yml` in order to add applications and be ready to get metrics
7. Add docker `daemon.json` file in `/etc/docker` in order to expose docker metrics for prometheus
   (a) Restart docker on app server in order to expose metrics

D.6.3 Add monitoring code to backend:

Added:

```
dotnet add package OpenTelemetry.Exporter.Prometheus.AspNetCore --version 1.4.0-rc.4
dotnet add package OpenTelemetry.Extensions.Hosting
dotnet add package OpenTelemetry.Instrumentation.AspNetCore
```

Added to Program.cs:

```
builder.Services.AddOpenTelemetry()
.WithMetrics(b => b.AddPrometheusExporter());

app.UseOpenTelemetryPrometheusScrapingEndpoint();
```

D.6.4 Add monitoring code to frontend:

Adding monitoring to the frontend entailed the exact same steps as adding monitoring to the backend. The following code got added to:

Minitwit.csproj

```
<PackageReference Include="OpenTelemetry.Exporter.Prometheus.AspNetCore" Version="1.7.0-rc.1
<PackageReference Include="OpenTelemetry.Extensions.Hosting" Version="1.7.0" />
<PackageReference Include="OpenTelemetry.Instrumentation.AspNetCore" Version="1.7.1" />
```

Program.cs

```
builder.Services.AddOpenTelemetry()
    .WithMetrics(b => b.AddAspNetCoreInstrumentation()
                       .AddPrometheusExporter());

app.UseOpenTelemetryPrometheusScrapingEndpoint();
```

D.6.5 Fix unresolved issues from last week:

D.7 Fix implementation of database abstraction layer so integration tests does not fail:

Integrations tests are failing for the implementations. Might be because the sql statements should be changes to another sql language.

D.8 Fix integrations tests:

D.9 Fix testing workflow visualisation in github actions:

D.9.1 Refine logging:

This task was downprioritized during the week, but these initial links for inspiration has been found.

`https://learn.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-7.0#log-level` `https://learn.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-7.0#log-category` `https://stackify.com/csharp-logging-best-practices/`

D.10 Week 7 - 15/03/24

Tasks: Add logging. Provide feedback for group f. Add configurations management.

Steps taken:

D.10.1 Provide feedback for group f:

- Do you see a public timeline? Yes
- Does the public timeline show messages that the application received from the simulator? Hmm. I see messages on the public timeline, but all the dates are from 2009 and i figure that if they were created by the simulator, the dates would be newer?
- Can you create a new user? Yes

- Can you login as a new user? Yes
- Can you write a message? Yes
- After publishing a message, does it appear on your private timeline? Yes
- Can you follow another user? Yes, and afterwards i see the user's messages on my private timeline.

Very good job all in all!!

D.10.2 Add logging:

Adding log messages to the code:

- In the controllers in Minitwit and SimulatorApi statements like the one below have been added where it makes sense:

```
_logger.LogDebug($"FollowUnfollowUser returns Forbid because user is not logged in");
```

D.10.3 Setting up Loki and Promtail:

We followed this youtube video: `https://www.youtube.com/watch?v=pnycjg_9M-o&t=664s&ab_channel=Thetips4you` that explained how to set up Loki and Promtail to grafana alongside this link: `https://ghazanfaralidevops.medium.com/grafana-loki-promtail-complete-end-to-end-project-d69`

- The youtube video linked some already made config files for Loki and Promtail that we could use, where we then changed to the right names end urls for our system. We added the two new config files to the monitoring server and added Loki and Promtail in the `docker-compose.yaml` file.
- For listening on our web server, we added the Promtail config file there as well and added Promtail to the `docker-compose.yaml` file in that server as well.
- At first we weren't getting the correct logs written by Amalie, that we knew should be displayed. We found out that this was due to following the tutorial too closely. Instead of getting our logs from /var/log/*, we had to get them from /var/lib/docker/containers. Adding the right path was the fix to our problem and we could immediately see the correct logs appearing in Grafana.

D.11 Adding configurations management:

D.11.1 Setup an Ansible control node

- Moved the vagrantfile that creates the web, and api application into the .infrastructure

- Made a new vagrantfile in the parentdir where the configuration management server would be set up
- In here we add installation commands to install the `ansible` package. This package comes with pre installed collections that contains different modules which we can later use in out ansible playbooks.

D.11.2 Make file to config management server called do_ssh_key

First I added the following line to `.bashrc`

```
export CONF_DO_TOKEN=<some token>
```

Then I added the following to the

- To upload the ssh key to digital ocean it needed the doctl which was first installed using snap
- Then the server generates a keypair without a passphrase
- We add the conf token to the server s.t. it knows which project we're working with
- Then we have to check whether DO already has a key of this name. If it does, we remove it
- We add the previous public key to DO

D.11.3 Create and push Ansible playbooks

Our strategy is to create two initial playbooks. One for our monitoring server and one for our web server. The documentation in Ansible and vagrant only specifies that the vagrant file that defines our web and monitoring server only needs to be referenced in the vagrant file as a new provisioner. We want the provisioning script in our root vagrantfile that creates the configuration management server to run vagrant up again and create the other servers. Here are the steps:

- Add playbook files for web and monitoring in their respective `.infrastructure` server directories.
- Alter the names of the servers in the Vagrant file to have test in them
- Create the configuration management server again
- Run vagrant up in the configuration management server
  - It halts and waits forever when provisioning

- Remove the hosts in the Ansible playbooks
- Run `vagrant up` again from the configuration management server
  - Still halts and waits forever
- Getting tired and frustrated at vagrant... having a discussion with the group about vagrant the following friday

D.12 Week 8 - 22/03/24

Tasks: Add three different linters. Setup Code Climate. Setup SonarQube. Add UI testing. Add Integration testing. Add end to end testing.

Steps taken:

D.12.1 Continuation of setting up Ansible

After having a talk with the other group members, we decided to not create a solution with ansible that collaborates with Vagrant. After some recent frustrations, we will look into which tool to use in the future to create the infrastructure. The steps taken here is to create a way to make ansible work in a github workflow and prepare the new infrastructure when we have to change accounts due to money expiration issues:

- By looking at the guide at github about creating a custom github action we can do it with a docker container.
- Create a new test repo for testing provisioning of VMs using github workflows
  - Include existing docker compose files and scripts for the two categories of VM's
- Create a docker file for the custom github action which should contain Ansible
- Add a requirements file to ensure that we install specific versions of collections
- Build and run docker image locally with

```
docker build -t ansible:latest .
docker run -it --rm ansible:latest bash
```

  - Run `ansible --version` to see that ansible is indeed installed
  - Run `ansible-galaxy collection list` to see that the collections is installed
- Create a make file for local development of the Dockerfile that contains two sections, namely build and run which performs the two commands stated above.
- Create github workflow to build and push the docker image to our private repo

- In order to create the custom github action we do the following:
  - Create directory at `.github/actions/<custom-action-name>`
  - Create a file called `action.yaml`
  - Create a new dockerfile that builds on top of the docker image we have created. It only needs to create an entrypoint that executes the `[entrypoint.sh](http://entrypoint.sh)` script
  - Create the `[entrypoint.sh](http://entrypoint.sh)` script which contains a call to the `ansible-playbook` command to run the playbook. The playbook will be provided as a argument defined by the action.yaml.
- Create a new github workflow that uses the custom github action. We do so by referencing the path

```yaml
- name: Provision and deploy web servers
  uses: .github/actions/ansible-run-playbook
```

- Add parameters to the github action

```yaml
inputs:
    dotoken:
        description: 'The Digital Ocean token'
        required: true
    dosshkeyname:
        description: 'The name of the ssh key in Digital Ocean'
        required: true
    sshkey:
        description: 'The private ssh key to access servers'
        required: true
    dockeruser:
        description: 'The username for the docker registry'
        required: true
    dockerpassword:
        description: 'The password for the docker registry'
        required: true
    inventoryfile:
        description: 'The working directory to run the command in'
        required: true
    playbook:
        description: 'The playbook to run'
        required: true
    working-directory:
        description: 'The working directory to run the command in'
        required: true
```

```
runs:
using: 'docker'
image: 'Dockerfile'
args:
    - ${{ inputs.sshkey}}
    - ${{ inputs.dockeruser }}
    - ${{ inputs.dockerpassword }}
    - ${{ inputs.dotoken }}
    - ${{ inputs.dosshkeyname }}
    - ${{ inputs.inventoryfile }}
    - ${{ inputs.playbook }}
    - ${{ inputs.working-directory }}
```

- Add arguments in the workflow with the `with` keyword
- Run the workflow
- For the web playbook we gradually added more modules to it:
  - Add firewall rules to the playbook with the `community.general.ufw` module
  - Run the workflow
  - Add folder synchronization with the `ansible.posix.synchronization` module for the web-playbook
  - Run the workflow
  - Add login and logout of docker
  - Run the workflow... fails
    - Add arguments in the `[entrypoint.sh](http://entrypoint.sh)` script to include docker username and docker password
  - Run the workflow
  - Add `docker_compose_v2` module to run docker compose up
  - Run the workflow... fails - Could not find the module
    - Ensure that the module is typed correctly - running workflow again fails
    - Try install `ansible-core` that has no collections installed to be sure that it uses the collection installed through the `requirement.yaml` - running workflow again fails
    - Try move the installed modules to a known directory - running workflow again still fails with cannot find the module
    - Reference the collection in the web-playbook.yaml - running workflow again stil fails with cannot find the module:

```
collections:
    community.docker:<version>
```

    * Rollback to use `ansible` instead of `ansible-core` and replace the docker compose module with a `shell` module and run `docker compose up -d`
        · This works!!!!!

- Now we copy paste the web-playbook to the monitor-playbook and change the settings to fit its state
- Run the playbook by adding it in the workflow - and it works!
- Move the github custom action related stuff to a new repo called `ansible-action`
- Create the new provisioning workflow in `itu-minitwit` but to not add any particular triggers since we will use it once we shift to new infrastructure
    - We reference the custom github action from the repo `ansible-action` by doing multiple checkouts and reference to the path it is checked out to:

```yaml
- name: Login to GitHub Container Registry
  uses: docker/login-action@v3
  with:
    registry: ghcr.io
    username: ${{ secrets.DOCKER_USER }}
    password: ${{ secrets.DOCKER_PWD }}

- name: Checkout repository
  uses: actions/checkout@v4
  with:
    path: main

- name: Fetch Ansible Playbook action
  uses: actions/checkout@v4
  with:
    repository: devops2024-group-e/ansible-action
    sparse-checkout: |
      .github/actions/ansible-run-playbook
    sparse-checkout-cone-mode: false
    path: action

- name: Provision and deploy web servers
  uses: ./action/.github/actions/ansible-run-playbook
  with:
    dotoken: ${{ secrets.DIGITAL_OCEAN_TOKEN }}
    dosshkeyname: ${{ secrets.DIGITAL_OCEAN_ANSIBLE_SSH_KEY_NAME }}
    sshkey: ${{ secrets.ANSIBLE_SSH_KEY }}
    dockeruser: ${{ secrets.DOCKER_USER }}
    dockerpassword: ${{ secrets.DOCKER_PWD }}
```

```
inventoryfile: inventory.ansible
playbook: web-playbook.yaml
working-directory: main/.infrastructure
```

D.12.2 Add three different linters:

Installing precommit:

```
brew install pre-commit
pre-commit sample-config > .pre-commit-config.yaml
dotnet tool install -g dotnet-format --version "7.*" --add-source https://pkgs.dev.azure.com
```

We also add the different linters to the config.yaml file.

We now want to add `hadolint`. We do so first by adding the hadolint to the `.pre-commit-config.yaml`.

Then we try to run pre commit with `pre-commit run -a` and could see that hadolint was not installed.

Now we install hadolint with `brew install hadolint`.

Now we add the `dotnet-format` linter to the pre commit config

We ran the pre-commit hooks but the dotnet formatter failed. So we installed dotnet sdk 8 and ran it again. Now it works!

In order to have hooks that run automatically when we commit, we run `install pre-commit`.

D.12.3 Setup code climate:

- Registred to code climate and added our repository
- `https://codeclimate.com/github/devops2024-group-e/itu-minitwit`

D.12.4 Setup SonarQube:

In order to analyze the software quality and technical debt of our project, we had to set up SonarQube as one of the tasks for the project work this week.

This was already a part of the given exercises, which meant that Helge had specified a guide in the lecture material on how to assemble it. Here he recommended the groups to use SonarCloud, which is another product from the same company behind SonarQube. The difference between the two services is that SonarCloud is a cloud-based service that easily integrates with GitHub.com, whereas SonarQube requires installation and maintenance on our server.

When registering on SonarCloud, one is able to use their GitHub login, which also makes adding the necessary repositories easier. You're then able to start add a new project by pressing the '+'

button found in the top right corner and pressing '*Analyze new project*', where you then have to give the application access to the specified repositories, which in our case is the minitwit repository.

Setting up your project will require you to decide on how to define what is considered new code. I here chose that this should be based on number of days, which is currently set to 30. This means that any code written in the last 30 days is considered new code. This is the recommended option for projects with continuous delivery like ours. The number of days can be changed if the current variable isn't suitable for our needs.

You are given two main options for setting up the actual analysis. You can choose methods such as GitHub Actions and other CI tools or you can choose automatic analysis if your used programming languages are supported by SonarCloud. This step wasn't specified in Helge's tutorial, so I chose to go with the automatic analysis, which was also the recommended option.

After this, the setup was complete and you can now see the measures of reliability, security and maintainability to mention a few. SonarCloud will from this point run on its own and start analyzing pull request when opened and any new code added to the main branch. The link can be found on the front page of our Notion.

After the set up was complete, I tried to figure out how to add code coverage since this wasn't added automatically. What I found out in my research was that code coverage wasn't automatically a part of the analysis when using the automatic analysis and you would have to introduce a separate third party tool, which would then send the information to SonarCloud to be analyzed. However, it does seems to be available when setting up the analysis with GitHub actions.

I discussed this with some of the members of the team and we decided to let it be for now, since the original task was just to set up SonarCloud and to instead bring it up for discussion in our next meeting to see whether we feel like code coverage in SonarCloud is a necessity in our project work.

D.12.5 Add UI testing with Selenium:

First we want to add Selenium to the test project along with the package for the ChromeDriver so we can do tests in Chrome browser. We also add WebDriverManager, as that seems important for getting the webdrivers to work in github actions, as well as WaitExtensions, so we can specify wait time when waiting for pageload etc.

- Add Selenium to test project with following commands:
  - `dotnet add package Selenium.WebDriver`
  - `dotnet add package Selenium.WebDriver.ChromeDriver`

- `dotnet add package WebDriverManager`
- `dotnet add package Selenium.WebDriver.WaitExtensions`

Then a folder is added to the Testing project named `UITests`, where standard testing files are added. We have added three testing files:

- `LoginRegisterLogoutUITests.cs`
- `FollowUnfollowUITests.cs`
- `MessagesUITests.cs`

All three initialise a Chrome driver and make it navigate to the localport, where the frontend is running, before executing the tests.

When the tests have been written the UI testing setup is also included in github actions by making sure to add a step in the `Frontend-Tests` in the `Testing.yml` that runs the PR-frontend, such that the tests can find the port they are testing on.

D.12.6 Add integration testing:

D.12.7 Add end to end testing:

D.12.8 Fix timestamps indatabase

First we want to create a test environment such that we can try and do migrations in our own local setup. We do so by dumping the data we currently have in the database on our docker container as the following:

```
pg_dump -U minitwit_sa -f dump.sql --column-inserts --data-only minitwit
```

We could have done this in a smarter way but now we want to extract that dump file from the container:

```
docker exec -it database /bin/sh -c "cat dump.sql" > dump.sql
```

And then we want to copy that file to our local machine with the following command(executed from our local machine, which has a ssh config with a reference to our prod server called devops):

```
scp devops:/minitwit/dump.sql ./dump.sql
```

Now because i have to try out different things, i have created a make file with two "commands" namely `db-up` and `db-down`. This is to quickly refresh the database. For that reason a `db-refresh` is also created that leverages these two commands:

```
.PHONY: db-down
db-down:
    docker exec -it minitwit-dev-database /bin/sh -c "psql -U minitwit-sa minitwit -c 'TRUNC

.PHONY: db-up
db-up:
    docker cp ./dump.sql minitwit-dev-database:/db-setup/dump.sql
    docker exec -it minitwit-dev-database /bin/sh -c "psql -U minitwit-sa minitwit < /db-set

.PHONY: db-refresh
db-refresh: db-down db-up
```

Now i can begin the work. First i look if we can just change the type in our table `message` from `integer` to `bigint`. We do so with the following query:

```
ALTER TABLE message
    ALTER COLUMN pub_date TYPE bigint;
```

That was allowed by the local test database without any issues. And since the datatype was a `integer` before it has converted it to a `bigint` before, because it has the same values a before.

Now I change the type in our schema file `schema.sql`. And then i change the datatype in the class `Message` to `long`.

Now we want to display it in a beautiful format. We start by doing the following:

- Create a Viewmodel for messages where the type for a timestamp is `DateTime`.
- Then we Convert the types we get from SQL to the viewmodel
- In the view we can then format the string by using `ToString("<Format for datetime string>")`
- We tried to use the `DisplayFormatAttribute` with `Html.For` in the view but resulted in a wrong format exception all the time. So we kept the original solution with `ToString()`.

D.12.9 Analyse and fix the slow response time of the register endpoint

We have found that the password hash makes up the majority of the time it takes to add a new user to the system:

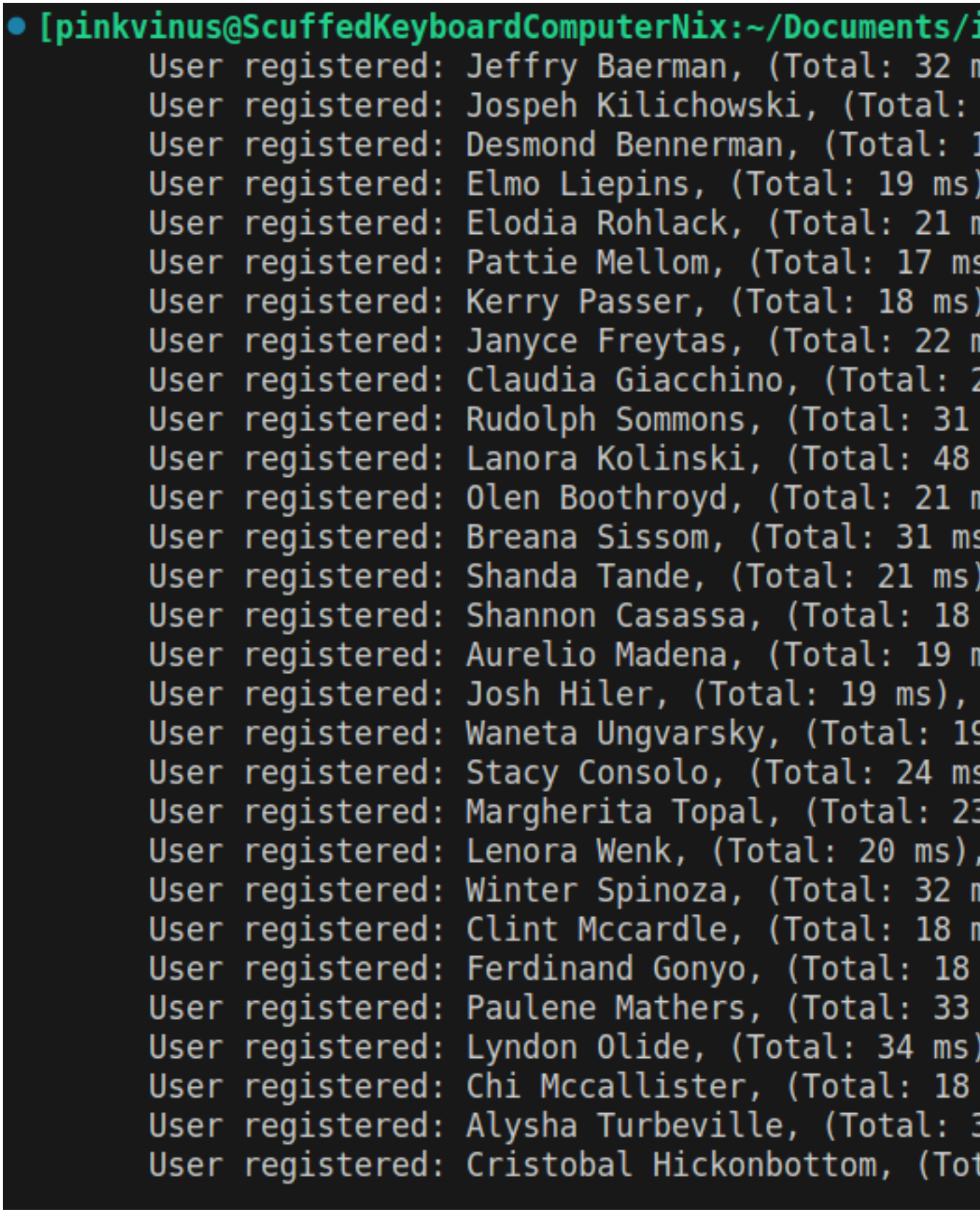

Figure 11: *Untitled*

Reducing the number of iterations (from 600000 to 6000) results in a much reduced response time

Changing the hashing algorithm to SHA512 and changing the number of iterations from 600000 to 210000 as recommended for this algorithm we yields a greater variance in response time, meaning that the fastest responses are faster, and the slowest responses are bout as slow as before. However, due to the fact that we need to be able to hash our already stored passwords, we will have to create a switch in the code, s.t. we would be able to introduce this new password hashing algorithm.

D.13 Week 9 - 05/04/24

Tasks: Add scaling to the project, migrate database. Add pre-commit to devcontainer.

Steps taken:

D.14 Adding pre-commit to devcontainer

This is a task from last week that we didn't manage to finish, as it turned out to be more complicated than we first thought. It has been solved by creating a dockerfile in the devcontainer that install first .net sdk 8, python, hadolint and pre-commit.

Problems i ran into:

- I need both sdk 7 and 8, because 7 are used for developing our application since they run in dotnet 7. However, pre-commit uses .net 8 to call the .net formatter. For that reason we need to install sdk 8 in the devcontainer. We tried to use the .net install script but it installed the sdk in a wrong location, so then we used the `--install-dir` flag to specify the location to install the SDK but for some reason that didn't work either. So we instead installed it at another location and copied the files to the existing installation of dotnet. The script looks like the following:

```
#!/bin/bash

wget https://dot.net/v1/dotnet-install.sh -O dotnet-install.sh
chmod +x ./dotnet-install.sh
./dotnet-install.sh --version latest --install-dir "/usr/share/dotnet-new"

sudo cp -u -r /usr/share/dotnet-new/* /usr/share/dotnet/
```

- I could tell that my team members couldn't use the devcontainer because they have another cpu, so i found a command that could fix this. The command `dpkg --print-architecture` finds the correct cpu architecture such that the image can run on different systems.

```
wget -q -O /bin/hadolint "https://github.com/hadolint/hadolint/releases/download/v2
```

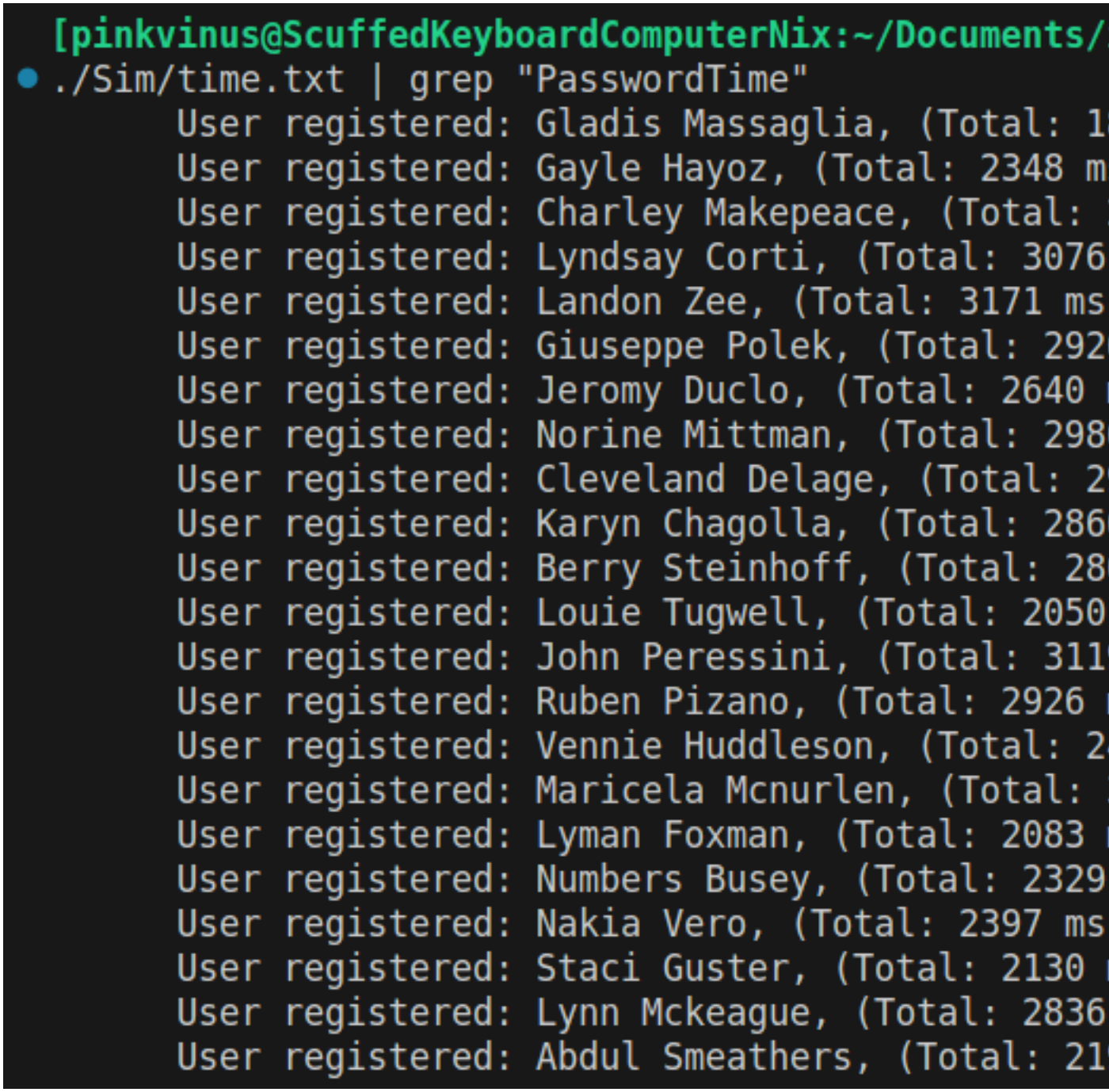

Figure 12: *Untitled*

- I didn't have the permission to use hadolint, so i gave the permission in the image with the following command:

```
chmod +x /bin/hadolint
```

D.15 Adding scaling to the project

- Useful resources
  - https://docs.ansible.com/ansible/latest/collections/community/docker/docker_swarm_module.html
  - https://labouardy.com/setup-docker-swarm-on-aws-using-ansible-terraform/
  - https://docs.docker.com/engine/swarm/swarm-tutorial/create-swarm/
  - https://www.digitalocean.com/community/tutorials/how-to-create-a-cluster-of-docker-con
  - https://www.pulumi.com/registry/packages/digitalocean/api-docs/droplet/

  How to setup Docker swarm with ansible:

  - Part 1
  - Part 2

We created a new server in the program

```
$ apt install docker.io
# the ip address here is the private ip for the manager server. (found on digital ocean)
$ docker swarm init --advertise-addr  192.168.1.6
Swarm initialized: current node (qvekvmy3xdpftrheics2kt05v) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-5bsq5avkideo5oqx6n4b5oth1nhqyg96msjp2igiome125afai-8x

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instruc
```

Now we have a swarm manager and a single worker in the swarm

We have found that the lack of documentation has made it very difficult to start working with this task.

D.16 Adding update strategy

We followed the tutorial linked to us in the lecture slides: https://docs.docker.com/engine/swarm/swarm-tutorial/rolling-update/

Here we quickly realized that in order to implement an update strategy, we would need a deployed service, which was a part of the 'adding scaling to the project' task. We therefore decided to leave this task until the deployment had been implemented and instead fixed the issue about adding UI test for the date formation.

D.17 Migrating Database

This is the second try, essentially, in migrating the database. We have now noticed a nice and useful guide on how to do it created by DigitalOcean.

So the first thing we do is to test it out. So we run our Pulumi workflow to setup our test environment. We then create some scripts to push the dump file that Andreas has of last weeks database to the test server by saying:

```
scp dump.sql devops-test:/minitwit/dump.sql
ssh devops-test

docker cp /minitwit/dump.sql database:/db-setup/dump.sql

docker exec database /bin/bash -c "psql -U minitwit-sa minitwit < /db-setup/dump.sql"
```

Now that the database has data, we begin using the guide from digitalocean. The setup current is the application server called `minitwit-test-web-02` and a managed database cluster with a database called `minitwit`.

According to the guide, we can create the following script:

```
#!/bin/bash

# First we need to run pg_dump for the old database server and save the dump file
echo "Fetch Database dump from source database"
pg_dump -h 164.92.253.253 -U minitwit-sa -p 5432 -Fc minitwit > ./minitwit-dump.pgsql

# Secondly we need to run pg_restore on the new database with the dump file that we have jus
echo "Restoring database on target database"
pg_restore -d <Connection uri string from DO> --jobs 4 ./minitwit-dump.pgsql
```

We noticed that if we had to run this on our local laptop then we had to allow the app server to have connections from the public on port `5432` so we ran the following command on the app server:

```
ssh devops-test
ufw allow in on eth0 to any port 5432
```

Another thing is that we had to create a file called `~/.pgpass`. Notice the squiqqly line. According to the postgres documentation it needs to be placed in there. We then add the following content in order to authenticate to the source database:

```
164.92.253.253:5432:minitwit:minitwit-sa:<Password of the db user>
```

Running the above resulted in an error saying that the `minitwit-sa` user does not exist. So we create that user in the DigitalOcean UI.

Now that we try to run it again, it fails with the following error:

```
pg_restore: error: could not execute query: ERROR:  relation "follower" already exists
Command was: CREATE TABLE public.follower (
    who_id integer NOT NULL,
    whom_id integer NOT NULL
);


pg_restore: error: could not execute query: ERROR:  permission denied for schema public
Command was: ALTER TABLE public.follower OWNER TO "minitwit-sa";

pg_restore: error: could not execute query: ERROR:  relation "latest" already exists
Command was: CREATE TABLE public.latest (
    id integer NOT NULL,
    command_id integer NOT NULL
);
```

Looking at this could mean that we should refresh that database. So we delete the database and add it again. We then run our script again, and it now fails with the permission denied.

Reading the documentation provided by docker they specify that in some situations you should run the commands with `no-owner` when doing `pg_restore`. This will not set the owner of the tables. Trying to run the script again removes the permission errors. However, we now encounter the following error:

```
pg_restore: error: a worker process died unexpectedly
```

By searching through the internet we see one at stackoverflow with a similar issue. We look at the version of postgres installed on our local laptop. But ours is `16.2` according to the `pg_dump --version` output. So we did the second option of removing the `--jobs 4` flag. We run it again. And it works!!!

We try to login to the server via pgadmin4 in context of the `minitwit-sa` which was not the user who created the tables and inserted the data. By doing some select statements on the tables we

could see that `minitwit-sa` does not have access. We give the `minitwit-sa` access to all tables in the public schema by executing the following query as `doadmin`:

```
GRANT SELECT, UPDATE, INSERT, DELETE ON ALL TABLES IN SCHEMA public TO "minitwit-sa";
```

This seems to work! Now the `minitwit-sa` can select the tables.

After trying to insert and delete entries with the `minitwit-sa` we saw that it does not have permission to lookup sequences to provide the next id for a user or anything else. So we execute the following command to fix this(as `doadmin` on the minitwit database):

```
GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA public TO "minitwit-sa";
```

The final script now looks like this:

```
#!/bin/bash

ssh devops "/bin/bash -c 'ufw allow in on eth0 to any port 5432'"

# First we need to run pg_dump for the old database server and save the dump file
echo "Fetch Database dump from source database"
pg_dump -h <web server ip> -U minitwit-sa -p 5432 -Fc minitwit > ./minitwit-dump.pgsql

# Secondly we need to run pg_restore on the new database with the dump file that we have jus
echo "Restoring database on target database"
pg_restore -d <DO URI connectionstring> --no-owner ./minitwit-dump.pgsql # look like postgre

psql -h minitwit-test-db-01-7035523-do-user-15869692-0.c.db.ondigitalocean.com -p 25060 -U d
psql -h minitwit-test-db-01-7035523-do-user-15869692-0.c.db.ondigitalocean.com -p 25060 -U d

ssh devops "/bin/bash -c 'ufw deny in on eth0 to any port 5432'"
```

D.17.1 Performing the migration

This is a TODO.

D.18 Save grafana dashboards and datasource files in repo

So since we are going to move our digital ocean account due to the money expiring, then it will have to move the board. This step here is going to be a simple one. We will have the files that is going to keep the layout. However, the solution will not provide automation to change the

endpoints of the datasource. But with this setup it should be easy to change a url and then push it again to see the live dashboards with data.

First we found the following site that shows how to use the Grafana provisioning feature. So we started by testing it out on a non-production environment. Since we already have a test monitor server, we are going to use that.

We create the same file structure and add the volume paths as it is shown in his Dockerfile.

We then export our existing dashboards to json files and save it in the `grafana/dashboard` directory.

After that we declare our datasource files. We can extract them as json by doing the following:

```
mkdir -p data_sources && curl -s "http://<ip of monitoring server>/api/datasources"  -u admi
```

We then copy them to the `provisioning/datasources` directory and change them to yaml.

Now on the test server we run `docker compose up -d` and check that it works. Both the datasources and dashboards is shown.

Now we copy the structure of the directory from the test server to the repo by using `scp` :

```
scp -r devops-test-mon:/minitwit-monitoring/grafana .infrastructure/monitor-server/grafana
```

By adding it to the above directory then Ansible should copy those files to the server and use it to start the grafana container.

D.19 Continuation of fixing the slow response time of the register endpoint

We had to implement a switch between the former and the current hashing algorithm in order for the `CheckPasswordHash` method to work with both old and new users in the system. To solve this issue, we made the following chances to /MinitwitSimulatorAPI/Utils/PasswordHash.cs:

```
public static string Hash(string value)
{
    var salt = RandomNumberGenerator.GetBytes(16);
    return GeneratePasswordHash(value, salt, "SHA512");
}
```

Here we added the string “`SHA512`” to the values being returned to help us with the switch in the next method.

```
private static string GeneratePasswordHash(string password, byte[] salt, string algorithm)
{
```

```
        string hashedResult;
        string result;
        switch (algorithm)
        {
            case "SHA512":
                hashedResult = Convert.ToHexString(KeyDerivation.Pbkdf2(
                    password: password,
                    salt: salt,
                    prf: KeyDerivationPrf.HMACSHA512,
                    iterationCount: 21000,
                    numBytesRequested: 32));
                result = $"SHA512${Convert.ToHexString(salt)}${hashedResult}";
                break;
            default:
                hashedResult = Convert.ToHexString(KeyDerivation.Pbkdf2(
                    password: password,
                    salt: salt,
                    prf: KeyDerivationPrf.HMACSHA256,
                    iterationCount: 600000,
                    numBytesRequested: 32));
                result = $"{Convert.ToHexString(salt)}${hashedResult}";
                break;
        }
        return result;
    }
```

We now check in the switch statement whether we got the "SHA512" string from the Hash method.

If we do, then we run the current hash algorithm, where we have lowered the number of iterations to 21000, which is a tenth of the recommend number from OWASP. We also add "SHA512" to the beginning of the password, which is needed in the CheckPasswordHash method in order to differentiate between the two versions of the algorithm.

If we didn't get the "SHA512" string, then we run the former hash algorithm, which hasn't been changed and still has the recommended number of iterations.

```
    public static bool CheckPasswordHash(string password, string hashedPassword)
    {
        var passwordParts = hashedPassword.Split('$');

        string pp;

        switch (passwordParts[0])
```

```
        {
            case "SHA512":
                pp = GeneratePasswordHash(password, Convert.FromHexString(passwordParts[1]), "SH
                break;
            default:
                pp = GeneratePasswordHash(password, Convert.FromHexString(passwordParts[0]), "SH
                break;
        }
        return pp == hashedPassword;
    }
```

We check in the switch statement whether the password include the "SHA512" string at the start in order to determine whether to run the current or the former hashing algorithm.

D.20 Week 10 - 12/04/24

Tasks: Switch Digital Ocean account, continue adding scaling to the project, add documentation and mapping of the system

Steps taken:

D.21 Finalization of fixing the slow response time of the register endpoint

To complete the task of fixing the slow response time in the register endpoint, we needed to create tests to make sure that the hashing algorithm worked as we had intended.

These tests are located in Minitwit.Simulator.Api.Tests/Controllers/RegisterControllerTests.cs

We ran Helge's minitwit simulator with both the former and the current version of the hashing algorithm in order to get hashed passwords that matched with users in the database. We then selected 3 different users with both versions of their hashed password in order to test that the correct usage of the passwords would return true and messing with their passwords would return false.

We had some trouble getting the PR for this issue accepted on GitHub due to pre-commit in the testing workflow raising an exception we couldn't replicate locally, but this was fixed by adding dotnet 8 to the pre-commit step.

D.22 Mapping the system

To map the system, we looked through our decision log to get an overview over the different tools we have introduced throughout the project. We also looked at Digital Ocean to get a sense of the servers and through our GitHub repository in order to recall details about how we set

up certain features. We decided to do an abstract overview of the system, as this was meant to help us keep track of how everything is connected and work with each other.

We added very short descriptions to each element in the diagram as separate text in order to give a quick overview. The decision log is meant for more lengthy paragraphs justifying our decisions, whereas this is meant for making sense of the system as a single unit.

D.23 Refactor FollowerController in SimulatorAPI

In order to get cleaner code we have refactored one of the methods in the FollowerController to receive follow/unfollow information via a method parameter instead of using a streamreader and other weird stuff.

D.24 Week 11 - 19/04/24

Tasks: Add one more frontend replica. Make security assessment. Handled a hacking attack!!

Steps taken:

D.24.1 Add one more frontend replica:

- In order for this to work we need the frontend to make use of sessions so that it still remembers who is logged in based on the id. Therefore we have added a session table in the database.
- We have added one more replica of the frontend such that there is now two of them.

We wanted to add one more frontend replica and that could be seen as a simple task, it sounds like we just need to add 2 instead of 1 in `.infrastructure/servers/swm-server/docker-compose.yaml`. The problem is that we use sessions for keeping track of the current users user_id. This session is in the memory of the frontend instance which means that the session will be lost when switching replicas. To solve this we started looking in to these two links that explained how to write code to be able to handle distributed sessions.

`https://learn.microsoft.com/en-us/aspnet/core/fundamentals/app-state?view=aspnetcore-8.0`

`https://www.nuget.org/packages/Community.Microsoft.Extensions.Caching.PostgreSql`

We started to add code for handling distributed sessions:

We added this package PostgreSQL in `Minitwit/Minitwit.csproj`:

```
<PackageReference Include="Community.Microsoft.Extensions.Caching.PostgreSql" Version="4.0.4
```

We added sessions settings in `Minitwit/project.cs`:

```
builder.Services.AddDistributedPostgreSqlCache(setup =>
{
    setup.ConnectionString = builder.Configuration.GetConnectionString("MinitwitDatabase");
    setup.SchemaName = "public";
    setup.TableName = "session";
    setup.DisableRemoveExpired = false;
    setup.CreateInfrastructure = false;
    setup.ExpiredItemsDeletionInterval = TimeSpan.FromMinutes(15);
});

builder.Services.AddSession(options =>
{
    options.IdleTimeout = TimeSpan.FromMinutes(5);
    options.Cookie.HttpOnly = true;
    options.Cookie.IsEssential = true;
});
```

and added a session table in `schema.sql`:

```
create table if not exists "session"
(
    "Id" text COLLATE pg_catalog."default" NOT NULL,
    "Value" bytea,
    "ExpiresAtTime" timestamp with time zone,
    "SlidingExpirationInSeconds" double precision,
    "AbsoluteExpiration" timestamp with time zone,
    CONSTRAINT "DistCache_pkey" PRIMARY KEY ("Id")
);
```

We then pulled it in to main without any errors or problem from the testing. When we went to the website and tried to log in, now with two replicas, we noticed that when we did log in we could see the text "You're now logged in" but we still were not properly logged in and couldn't do any of the task we usually can. We could therefore see that the distributed sessions isn't working as they should.

First we checked if the database was correctly connected and could see that it wasn't. So we fixed a proper connection but still had the same problems.

We also tried to set upp the distributed sessions with redis as image in `.devcontainer/docker-compose.yaml`

```
    - main
environment:
```

```
      - Minitwit_ConnectionStrings__MinitwitDatabase=Host=minitwit-dev-database;Port=5432;Userna
      - Minitwit_ConnectionStrings__Redis=minitwit-cache:6379

  minitwit-db:
    build:
@@ -27,3 +28,11 @@ services:
      - main
    ports:
      - '5432:5432'

  cache:
    image: redis:7.2.4
    container_name: minitwit-cache
    networks:
      - main
    ports:
      - '6379:6379'
```

And added arguments for the secrets in `.github/workflows/Provision-and-Deploy.yaml` file:

```
    build-args: |
            DB_CONN=${{ secrets.DB_CONNECTION }}
            REDIS_CONN=${{ secrets.REDIS_CONNECTION }}
```

and testing if it's the image is deployed in.`github/workflows/Testing.yaml`:

```
     - name: Set up redis cache
       run: docker compose -f docker-compose.yaml up cache -d
```

Specified it in `.infrastructure/servers/swm-server/docker-compose.yaml`:

```
      cache:
      image: redis:7.2.4
      networks:
        - main
      ports:
        - '6379:6379'
```

In `Dockerfile.frontend` we added:

```
    ARG REDIS_CONN=""
```

and:

```
ENV Minitwit_ConnectionStrings__MinitwitDatabase=${REDIS_CONN}
```

In `Minitwit/Minitwit.csproj` we added two packages:

```
<PackageReference Include="Microsoft.Extensions.Caching.StackExchangeRedis" Version="7.0.18"

<PackageReference Include="StackExchange.Redis" Version="2.7.33" />
```

And in the `Minitwit/Program.cs` we changed up the code for how to handle the cached sessions:

```
builder.Services.AddStackExchangeRedisCache(options =>
{
        options.Configuration = builder.Configuration.GetConnectionString("Redis");
options.InstanceName = "minitwit_";
});

builder.Services.AddSession(options =>
{
    options.IdleTimeout = TimeSpan.FromMinutes(5);
    options.Cookie.HttpOnly = true;
    options.Cookie.IsEssential = true;
});
```

In the `Minitwit/appsettings.Development.json` file we specified the connection string and the port:

```
"ConnectionStrings": {
    "MinitwitDatabase": "Host=127.0.0.1;Port=5432;Username=minitwit-sa;Password=123;Database
    "MinitwitDatabase": "Host=127.0.0.1;Port=5432;Username=minitwit-sa;Password=123;Database
    "Redis": "localhost:6379"
}
```

and lastly we added this code to `docker-compose.yaml`:

```
  cache:
    image: redis:7.2.4
    container_name: minitwit-cache
    networks:
      - main
    ports:
      - '6379:6379'
```

Figure 13: *Untitled*

After this we pulled to main again and tried to log in to Minitwit again and could then see that it still didn't work and that we had the same problem as before, that we got the logged in message but still wasn't properly logged in. After all the time we put on trying to make this work we decided that the problem itself isn't so essential that we should put anymore time on it. We therefore revoked all of the changes we previously made and went back to having one replica for the frontend instead.

D.24.2 Make security assessment:

The security assessment can be seen in a separate document.

D.24.3 Got hacked and handled it:

We got hacked as described in another document. We deleted our docker swarm servers with Pulumi to get rid of it.

D.25 Week 12 - 26/04/24

Tasks: Harden the system based on our security assessment from last week. Prepare diagrams of the system.

Steps taken:

D.25.1 Harden the system based on our security assessment from last week:

- Close ports:
  - We started by using *Kali Linux* to run **nmap**, which told us of the following open ports in our system: The 22/tcp is for *ssh*, the 80/tcp and 8080/tcp are for *http* and the 9090/tcp is for *Prometheus*, where the latter doesn't require an open port. To fix this, we instead exposed the port to a private network via the interface eth1 and denied access via eth0.

When we went into the shell of the server, we could see that the firewall was working as expected, but we were still being told by nmap that port 9090 was open. After trying to remove all access to port 9090 and not seeing any changes in Kali Linux, we went back to the solution described above and left if for Friday to investigate further.

- We then ran **OWASP zap** to scan our website for vulnerabilities in regards to the ports. This showed us some warnings about not being fully protected from injection attacks or attacks made with the header, but none about the ports, so we didn't make any actions from this.
- Lastly, we wanted to upgrade the system from HTTP to **HTTPS**, as the former is not secure and this would likely solve some of our security issues. To do this, we tried to follow the guide made by Mircea from the 11th lecture:
  - First attempt
    1. Create a domain -> We bought the domain grlpwrtwit.dk on simply.com
    2. Set the name servers -> This is where we ran into a lot of issues. Whenever we tried to change the name servers on simply.com, we got an error message saying to control what we wrote. After doing some digging, we found a post that said Punktum.dk was where we had to go to change the name servers due to our domain ending in .dk. Again, we tried to follow the online tutorials, but we didn't have access to the correct link as shown in the guides. Creating tickets and trying to reach out to the support on simply.com didn't help either.
  - Second attempt
    1. Create a domain -> We bought the domain grlpwrtwit.se on Loopia.se
    2. Set the name servers -> We here had to first disable DNSSEC and then we were able to change the name servers to ns1.digitalocean.com and ns2.digitalocean.com as told by the guides.

- Prevent injection via logging:
  - The user inputs have been sanitised in order to prevent injection attacks. Before the user input was put directly into logging. We have now replaced newlines `\n` and carriage returns ” with '_'.

- Update dependencies in pipeline:
  - Enabled security checks and package updates by dependabot in github repo security settings
  - Added `dependabot.yml` config file to workflow, which checks Github Actions and NuGet packages for outdated dependencies.
  - Dependabot automatically adds PRs to the project for outdated packages

E Security Assessment

E.1 Risk Identification

E.1.1 Asset Identification

By browsing through our setup and documentation, we identified the following list of assets:

- Web application
- Public GitHub repository
- Digital Ocean servers
- Tools
  - Ansible
  - Code Climate
  - Docker, Docker Compose & Docker Swarm
  - GitHub Actions
  - Grafana
  - Linters
  - Loki
  - Pulumi
  - SonarCloud

E.1.2 Threat Source Identification

To help us identify possible threats to the system, we have consulted the OWASP *Top 10 Web Application Security Risks*[7], which describes the following threats:

1. Broken Access Control
2. Cryptographic Failures
3. Injection Attacks
4. Insecure Design
5. Security Misconfiguration
6. Vulnerable and Outdated Components
7. Identification and Authentication Failures
8. Software and Data Integrity Failures

9. Security Logging and Monitoring Failures
10. Server Side Request Forgery

E.1.3 Risk Scenario Construction

Based on the information gathered from the two previous steps, we have constructed the following risk scenarios and outlined which of the OWASP top 10 risks that would affect the described scenarios:

1. **URL Tampering**

   The attacker can construct a URL in the **web application** with a user ID, such that they can bypass login and are able to write a message from another user's account.

   This would be an issue of *Broken Access Control* and *Server Side Request Forgery*.

2. **Log Injection**

   The attacker can fabricate log information via an injection attack in the **web application** as a means to hide their activity and ill-intentioned actions.

   This would be an issue of *Security Logging and Monitoring Failures* and *Injection Attacks*.

3. **Password Brute Forcing**

   The attacker can brute force login credentials in the **web application** by taking advantage of no timeouts and a weaker hash implementation.

   This would be an issue of *Cryptographic Failures* and *Identification and Authentication Failures*.

4. **Depricated Dependencies**

   The attacker can identify weak, outdated or depricated tools and dependencies in the system's CI/CD pipeline via **GitHub Actions**, which is publicly availble through the **GitHub repository**.

   This would be an issue of *Vulnerable and Outdated Components* and *Software and Data Integrity Failures*.

5. **Open Ports**

   The attacker can scan the public IP addresses of the **Digital Ocean** servers to find unnecessarily or unexpectedly open ports with known vulnerabilities, which can be exploited in further attacks.

   This would be an issue of *Security Misconfiguration*.

6. **SQL Injection**

   The attacker can target the login page of the **web application** with SQL-injection attacks to strike the database.

   This would be an issue of *Injection Attacks*.

7. **Exposed Secrets**

   The attacker can get access to the system, or associated tools, via secrets found written in the code in the public **GitHub repository**.

   This would be an issue of *Identification and Authentication Failures* and *Security Misconfiguration*.

E.2 Risk Analysis

E.2.1 Likelihood Analysis

Likelihood will be graded on the following scale: {Rare, Unlikely, Possible, Likely, Certain} with *Rare* being the least severe and *Certain* being the most severe.

1. **URL Tampering**

   We examined the different URLs on the web application and could not find any where the IDs or login parameters were exposed. Therefore we deemed the likelihood to be **Unlikely**.

2. **Log Injection**

   We received warnings from SonarCloud that we had logging vulnerabilities in our code several different places. Therefore we deemed the likelihood to be **Likely**.

3. **Password Brute Forcing**

   We currently do not have any measures in place to combat brute force attacks and no password requirements for the users when signing up. Therefore we deemed the likelihood to be **Possible**.

4. **Depricated Dependencies**

   Our GitHub repository, and thereby our workflows for GitHub Actions as well, are public for anyone to see. In the workflows we are using templates of actions made available and written by others, alongside showcasing some of various tools we use. We currently rely on the fact that the actions and tools we use are secure, but have not incoporated anything that checks whether that is true. However, many of the tools we use are well-known and therefore we hope that known vulnerabilities are getting discovered rather quickly. Therefore we deemed the likelihood to be **Possible**.

5. **Open Ports**

   All of the IP addresses for our servers are public in Digital Ocean, which makes it very easy to scan for vulnerabilities. We have put up firewalls and have taken measures to only have necessary ports open, but are also aware that accidental port exposure, through some of the tools we are using, is possible. We are convinced that this is how our server got hacked during the course. Therefore we deemed the likelihood to be **Certain**.

6. **SQL Injection**

   We have made sure to sanitize user input on the login page of the web application. Therefore we deemed the likelihood to be **Rare**.

7. **Exposed Secrets**

   We have been conscious to ensure that secrets are either kept locally where only ourselves can access them, such as secret keys for logging into the servers, or used the "environment secrets" tool on GitHub, if secrets had to be accessed from the repository. Additionally, we have not made generic passwords, but rather used random password generators to get stronger passwords. Therefore we deemed the likelihood to be **Rare**.

E.2.2 Impact Analysis

Impact will be graded on the following scale: {Insignificant, Negligible, Marginal, Critical, Catastrophic} with *Insignificant* being the least severe and *Catastrophic* being the most severe.

1. **URL Tampering**

   This would breach both the confidentiality and the integrity of the system's security. We still keep all our data, though the data would have been compromised. Therefore we deemed the impact to be **Critical**.

2. **Log Injection**

   This could be used to disguise an attacker's activity and attack attempts on the web application. However, it would not give them access to the server or applicition itself, nor other data than the logs. We would still want to know if someone was trying to attack our system. Therefore we deemed the impact to be **Marginal**.

3. **Password Brute Forcing**

   This would breach both confidentiality and integrity. In very severe cases, it could also affect the availability, if the requests to login became too intense. We would still keep all of our data, though the data would have been compromised. Therefore we deemed the impact to be **Critical**.

4. **Depricated Dependencies**

   This would have a very big attack surface, as we most likely would not know which tool or where in the application process we could have a vulnerability. The target for a vulnerability could thereby vary in severity, but could in the worst case have severe consequences. Therefore we deemed the impact to be **Catastrophic**.

5. **Open Ports**

   This again would have a big attack surface, as we do not know which tool or where in the application we potentially could have a vulnerability exposing ports for an adversary to attack. If the attacker ended up gaining access to our servers, they would have full

control over the application in the worst case. Therefore we deemed the impact to be **Catastrophic**.

6. **SQL Injection**

   This would breach both confidentiality and integrity. The data could both be compromised and lost, but we would have a backup of the database. Therefore we deemed the impact to be **Critical**.

7. **Exposed Secrets**

   If any of the secret in the GitHub repository were to fall into an attacker's hands, they would be able to get access to our setup and thereby dismantle the entire system. Therefore we deemed the impact to be **Catastrophic**.

E.2.3 Risk Matrix

Based on the points made about the likelihood and impact of each risk scenario, we have constructed the following risk matrix to indicate the severities and prioritize the scenarios:

	Rare	Unlikely	Possible	Likely	Certain
Catastrophic	Exposed Secrets		Depricated Dependencies		Open Ports
Critical	SQL Injection	URL tampering	Password Brute Forcing		
Marginal				Log Injection	
Negligible					
Insignificant					

Figure 14: *Risk matrix based on security assessment*

E.2.4 Action Plan

We discussed the results of the security assessment and decided on focusing on the risk scenarios placed in the red area of the risk matrix.

For the open ports, we went through our firewall settings, which were handled by the Ansible ufw module. Here we scanned our main server's IP address with *nmap* to see the current open ports. We tried to close the port we had open for Prometheus, but got conflicting results when we checked the firewall itself compared to the scan of the IP address. The cause of this is expected to be Docker bypassing the firewall through IP tables.

For the log injection, we went through our code and ensured that all user data was sanitized before added to the log, such that it could not be tampered with.

For the depricated dependencies, we added the tool *dependabot* to our CI/CD pipeline, which makes sure that the dependencies used in our Minitwit system are up to date and thereby less vulnerable to older known exploits.

For the password brute forcing, we decided to leave it as it, due to the group not having enough information on how the creation of users and login works in the simulator and thereby not knowing if the simulator could handle password restraints or 2-factor authentication, which could have been our solution to improve this issue.